(2)

論理型言語向きプロセッサの アーキテクチャに関する研究

平成3年8月

中島　浩

目次

1 序論 1

2 論理型言語とその処理方式 7

2.1 言語の構造 9

2.2 ユニフィケーション 14

2.2.1 基本的なデータ表現 14

2.2.2 変数の表現 15

2.2.3 基本的なユニフィケーション 19

2.2.4 構造体のユニフィケーション 21

2.3 バックトラック 28

2.3.1 状態の保存 28

2.3.2 Undo と Trail 31

2.3.3 カット 32

2.4 基本的な最適化 37

2.4.1 Tail Recursion Optimization 37

2.4.2 Clause Indexing 40

2.5 ESP/KL0 43

2.6 KL1 45

3 推論マシンのアーキテクチャ 47

3.1 インタプリタ方式による逐次型推論マシン (PSI-I) 51

3.1.1 設計方針 51

3.1.2 機械命令アーキテクチャ 53

3.1.3 ハードウェア構成 56

3.1.4 レジスタ・ファイル 59

3.1.5 タグ・アーキテクチャ 63

3.1.6 メモリ・アーキテクチャ 70

3.1.7 マイクロ命令アーキテクチャ 74

3.2 コンパイル方式による逐次型推論マシン (PSI-II) 79
3.2.1 設計方針 79
3.2.2 機械命令アーキテクチャ 81
3.2.3 ハードウェア構成 96
3.2.4 命令フェッチ／デコード機構 100
3.2.5 タグ・アーキテクチャ 105
3.2.6 メモリ・アーキテクチャ 107
3.2.7 マイクロ命令アーキテクチャ 113
3.3 パイプライン方式による逐次型推論マシン (PSI-III) 119
3.3.1 設計方針 119
3.3.2 ハードウェア構成 123
3.3.3 パイプライン・アーキテクチャ 125
3.3.4 タグ・アーキテクチャ 133
3.3.5 メモリ・アーキテクチャ 141
3.3.6 マイクロ命令アーキテクチャ 143
3.4 並列推論マシン 145
3.4.1 システム構成 146
3.4.2 プロセッサ間通信機構 148

4 最適化手法 151
4.1 組込述語の最適化 153
4.1.1 組込述語の性質 153
4.1.2 引数の受渡し 156
4.2 バックトラックの最適化 159
4.2.1 Clause Indexing 161
4.2.2 Neck Cut Optimization 164
4.2.3 トレイル 173
4.3 例外処理の最適化 175
4.3.1 例外の検出 176
4.3.2 動的な呼出 178

4.4 複合命令 180

5 評価 183
5.1 実行速度 184
5.1.1 方式による性能差 184
5.1.2 他のマシンとの比較 193
5.2 最適化の効果 200
5.2.1 ベンチマークによる評価 200
5.2.2 実用プログラムによる評価 206
5.3 メモリ・アクセスの特性 207
5.3.1 PSI-I の評価 208
5.3.2 PSI-II の評価 212
5.3.3 PSI-III の評価 215

6 結論 219

謝辞 223

発表論文一覧 225

参考文献 227

付録 237
A ESP/KL0 237
A.1 オブジェクト指向機能 237
A.2 実行順序制御 246
B KL1 264
B.1 言語仕様 264
B.2 処理方式 268
C WAM 命令一覧 278
C.1 get 系命令 281
C.2 put 系命令 282

C.3 unify 系命令 282
C.4 control 系命令 284
C.5 indexing 系命令 284
D WAM の実行例 286

第1章
序論

計算機科学の発展に伴い，その応用分野は伝統的な科学技術計算や事務計算から，人工知能や知識情報処理といった人間の思考活動に近い分野に拡大しつつある。このような「知的な」応用のためには，処理可能なデータの量を単純に増やすことだけではなく，データの質的な向上が求められる。即ち，「概念」や「知識」のようなものを取扱うためには，数値，配列，文字列などの定型的なデータだけでは全く不充分であり，（少なくとも）木構造やグラフのような複雑な構造を持ったデータを処理することが必要となる。また，単に複雑であるばかりではなく，処理の進行に伴いデータの大きさや構造が変化することも，知的なデータ処理にとって欠かすことのできない特性であろう。

さて，このような複雑かつ動的に変化するデータを取扱うには，FORTRAN や C に代表される手続型言語では不便な点が多い。例えば二進木の一つのノードを表現するには，ノードに与えられたデータ，左側の枝，及び右側の枝の三つの要素が必要となる。FORTRAN の場合，三つの要素を持つ構造体を定義することはできるが，「枝」を直接表現する方法がないため，例えば構造体の配列の要素番号を用いるなどの，迂遠な方法を採らなければならない。C の場合には，ポインタ変数を用いることによってこの問題はある程度解決するものの，「データのアドレス」という内部表現に近い形での記述となってしまう。また，ノードにどのような「型」を持ったデータを与えるかを予め定めておかなければならないことも，データ表現の柔軟性を大きく阻害する要因となっている。

データ構造の表現方法だけではなく，ノードの付加や除去などの基本的な操作の記述も，これらの言語では必ずしも容易ではない。FORTRAN では動的な記憶領域の割付が記述できないため，取扱うことのできる二進木の大きさの上限を，配列の要素数などの形でプログラムの中に明示的に宣言する必要がある。C には記憶領域の割付や解放のためのプリミティブが用意されているが，効率的かつ正しい記憶管理を行うには，注意深いプログラミングが必要であることが多い。

以上のような欠点は，手続型言語のデータ表現が「データ型の宣言」に基いていることに起因している。即ち，取扱うデータの構造や大きさがコンパイル時に決定していることが原則となっているため，動的に変化するデータの表現や操作が極めて困難になっているのである。

一方，記号処理用言語と呼ばれるプログラミング言語では，動的に変化する複雑なデータを容易に取扱うことができることが，大きな特徴となっている。代表的な記号処理用言語であるLISP及びPrologでは，階層的な構造を持ったデータであるリストや複合項が処理の基本的な対象となっており，複雑なデータを極めて容易に表現し操作することができる。また，データの型はコンパイル時ではなく実行時に決定されるため，例えば「二進木の探索」はノードに付与されたデータがどのようなものであっても，同じプログラムによって実現することができる。更に，データの生成や消滅に伴う記憶領域の割付や解放は全て自動的に行われ，プログラマに対する記憶管理の負担は極めて小さいものとなっている。

さて，記号処理言語のこのような特徴は，「知的な」応用プログラムを作成するプログラマにとって極めて有益なものであるが，言語を処理する側，即ちコンパイラやインタプリタにとっては逆に大きな負担となっている。その中で最大のものは，データ型が実行時に決定されるという性質である。一般にプログラミング言語では，データのアクセスや演算などの操作の具体的な内容は，対象となるデータの型によって異なっている。例えば配列の要素を取り出す操作の場合，その対象が配列でなければエラーになる。また，配列の要素が整数であるのか，文字であるのか，または別の何かであるのかによって，要素を取り出す手順（即ちアドレス計算の方法）が異なるのが普通である。手続型言語の場合，操作対象が例えば整数の配列であることが宣言されるため，コンパイラが操作の正当性をチェックし，かつ要素の型に応じたアドレス計算のためのコードを生成する。従って，実行時には操作対象が何であるかを判断する必要は一切ない。

しかし記号処理言語の場合には，データ型が決定するのは実行時であるので，正当性のチェックや操作手順の選択といったデータ型の判断は，実行時に行わなければならない。このため記号処理言語におけるデータの内部表現には，データの「値」，即ち数値やアドレスなどの他に，データの型を示すためのタグが付加されている。従ってデータ型の判断は，タグの抽出とその値に基く処理手順の決定という操作によって実現されるが，これを汎用計算機上で行う場合，AND演算，比較演算，条件分岐など，いくつかの命令を実行しなければならない。しかもデータ型の判断は，データの参照，演算，更新といったごく基本的な操作に付随しているため，手続型言語に比べて処理速度が格段に低下してしまう結果となる。

この他，データの生成／消滅に伴う記憶領域の割付／解放のための操作や，頻繁に行われる関数／述語の呼出／復帰の操作など，プログラマにとって便利な機能が高速な処理を妨げる要因となっている。その結果，記号処理言語は「便利だが遅い」という不名誉な評価を受けることとなり，その利用分野が限定され，ひいては計算機の知的な応用に関する研究を阻害する一因ともなっていたのである。

このような状況を大幅に改善したのが，記号処理言語のための専用プロセッサの出現である。まず先鞭を付けたのはLISPプロセッサであり，1970年代から数多くのプロセッサの研究や試作が成され，その内のいくつかは商用化されるなどの成功を収めた[Yasui 82, Ida 85, Nakashima 90b]。LISPプロセッサの最大の特徴は高速性であるが，その鍵となっているのはハードウェアによる並行処理である。即ち，データ型の判定などLISPの処理に必要な「付随的な」操作を，特別のハードウェアを設けることによって，演算などの「本来の」操作と並行して行うことにより，オーバヘッドを大きく軽減することができたのである。

一方論理型言語であるPrologに関しては，LISPに比べて言語の歴史が浅いこともあって，専用プロセッサの研究はかなりの遅れをとっていた。また，ユニフィケーションやバックトラックというLISPよりも更に複雑な操作が処理の基本であるため，言語とハードウェアの間のセマンティクス・ギャップが大きいことも，研究を困難なものにしていた。しかし一方では，汎用計算機上での実用的な処理系の出現や，日本の第五世代コンピュータ・プロジェクトでの基本言語として採用されたことなどから，言語の利用者が急速に増加し，本格的な専用プロセッサの実現へ大きな期待が寄せられた。

本研究はこのような背景で成されたものであり，論理型言語のための専用プロセッサである推論マシンのアーキテクチャはいかにあるべきかを検討し，高速かつ実用的なプロセッサを実現することが最大の目的であった。そのためには，LISPプロセッサにおいて成功を収めたハードウェアによる並行処理の概念を，論理型言語の処理においていかに活用するか，即ちどのようなハードウェアを投入するかの選択と評価が重点的な課題となった。また，コンパイル技術などの言語処理技術が未熟であったため，様々な最適化手法を考案し，かつその効果を実証することも必要であった。更に，第五世代コンピュータ・プロジェクトの最終目標である並列推論マシンの実現のための研究も，重要課題の一つであった。

以上のような目的と課題に基き，本研究は第五世代コンピュータ・プロジェクトの一環として行われ，以下に示す成果を挙げることができた。

(1) 基本アーキテクチャの提案：
本研究を通じて三つの逐次型推論マシンPSI-I, PSI-II, PSI-IIIの設計／試作を行ったが*，これらのアーキテクチャに関する基本的な考え方を，世界初の推論マシンであるPSI-Iの設計において提案した。即ち；

(a) 様々なデータ型を表現する「タグ」操作のハードウェア
(b) 複数のスタックを効率的に実現する記憶管理ハードウェア
(c) これらのハードウェアを効率的かつ柔軟に制御する水平型マイクロプログラム

の三つの要素を，推論マシンと言う未知の計算機のアーキテクチャにおいて統合し，かつその有効性をPSI-Iの開発により実証したことは，論理型言語と計算機アーキテクチャの二つの研究分野に大きく貢献するものであった。

*PSI-I, PSI-II, PSI-IIIについては，それぞれ「逐次型推論マシンPSI」，「逐次型推論マシンPSI-II」，「PIM/m要素プロセッサ」という呼称が正式なものであるが，本論文では相互の関連性と相違を明確にするため，PSI-I, PSI-II, PSI-IIIと呼ぶこととする。

(2) インタプリタ方式とコンパイル方式の評価：
PSI-I の実行方式は，ソース・プログラムに類似した構造を持つ「内部表現」を，マイクロプログラムのインタプリタが解釈／実行するものであった。一方これとは対照的な方式として，WAM (Warren Abstract Machine) と呼ばれる抽象マシンの命令へコンパイルする方式が提案された [Warren 83]。筆者らはこの二つの方式の優劣を判定するために，PSI-I 上に WAM の命令セットのエミュレータを実験的に構築し，両者の性能を比較した。その結果，コンパイル方式ではインタプリタ方式の 2 ～ 3 倍の性能を得られることを明かにするとともに，性能差をもたらした要因がどのようなものであるかを詳細に解析した。このように，異る実行方式を同一のハードウェアを用いて定量的に評価する研究は，少なくとも論理型言語の分野では例がなく，WAM の命令アーキテクチャの評価に大きな役割を果たすことができた。

(3) コンパイル方式向きアーキテクチャの提案：
第二の逐次型推論マシン PSI-II は，上記の研究成果に基づいて設計されたものであり，コンパイル方式による推論マシンに関する先駆的役割を果たした。また，スタック・オーバーフローの検出機構，例外処理の割込による実現，組込述語に関する命令アーキテクチャなど，論理型言語に特有の処理機構についての様々な新しいアイデアを提案した。これらのハードウェア機構や処理方式の有効性は，430 KLIPS* という当時では世界最高の性能を達成したことにより実証され，後続の様々な推論マシンのアーキテクチャに大きな影響を与えた。

(4) 論理型言語向きパイプライン方式の提案：
汎用マシンにおける命令パイプラインの有効性は以前から実証されていたが，論理型言語の処理においても処理方式の特徴を生かしたパイプラインの導入が有効ではないかと考えた。そこで，処理の様々な局面で行われるデータ型の判定やデレファレンス†をパイプライン化し，ハードウェアの並行処理の範囲を更に広げる方式を考案し，第三の逐次型推論マシンである PSI-III に実装した。この方式は推論マシンのアーキテクチャとして画期的なものであり，1.5 MLIPS という極めて高い性能を達成する上で極めて重要な役割を果たした。

(5) 最適化手法の提案と評価：
論理型言語の歴史が比較的浅かったため，最適化に関する技術は十分に成熟したものではなかった。筆者らは三つの推論マシンの開発を通じ，様々な最適化手法，特に実用的なプログラムに適用でき，かつ高い効果が望める手法を考案した。具体的には，組込述語の引数受渡し，バックトラックに関する種々の処理，実行時に検出される例外の取扱いなどであり，いずれも従来にない新たな手法であった。また，これらを実際の処理系に組み込むとともに，高性能推論マシンの実現に大きく貢献していることを，詳細な評価によって明かにした。

*Kilo Logical Inference per Second の略であり，一秒間に行われるゴール呼びだし回数を意味する。
†操作対象のデータを得るために，任意長のポインタの連鎖を手繰る処理。

(6) 並列推論マシンの実現：
前述の三つの推論マシンは，全て並列推論マシンの要素プロセッサとしても用いられた。特に PSI-II を要素プロセッサとする Multi-PSI/v2 は，64 プロセッサという大規模な構成であるとともに，並列論理型言語 KL1 やオペレーティング・システム PIMOS を実装した本格的な並列プロセッサである。また，PSI-III を要素プロセッサとして用いることにより，Multi-PSI/v2 を性能と規模の両面で発展させた並列推論マシン PIM/m の開発も行った。これらの並列推論マシンの開発を通じて得られた知見，即ちストリーム通信の最適化，構造データの要素の更新，プロセッサ内外での実時間ガベージ・コレクションなどは，並列論理型言語の研究にも大きなインパクトを与えた。

以下，本論文では次のような構成にしたがって，論理型言語向きプロセッサのアーキテクチャについて論ずる。

まず第二章では，プロセッサのアーキテクチャを論ずるための基礎として，論理型言語がどのようなものであり，またどのように処理されるかを述べる。具体的には，代表的な論理型言語である Prolog について，その言語仕様や処理の方式を論ずる。また，筆者らが開発した逐次型推論マシンの基本言語である ESP/KL0，及び並列推論マシンの基本言語である並列論理型言語 KL1 についても，それぞれの特徴を簡単に述べる。

次に第三章では，本論文の主題であるアーキテクチャの問題を，筆者らが開発した幾つかの推論マシンを対象として論ずる。特にこれらのマシンにおいて，タグ操作や記憶管理などの論理型言語特有の処理のために，どのようなハードウェアが用意され，またどのように利用されるかを詳しく述べる。また，これらのマシンの相互の相違点を対比することにより，アーキテクチャの改良がいかなる観点から行われたかを明らかにする。

第四章では，論理型言語の最適化に関して，筆者らが独自に提案し実証した様々な手法について述べる。具体的には，まず論理型言語に特有の処理であるバックトラック関するものの他，実用的なシステムでは欠かすことのできない組込述語や例外処理の最適化についても論ずる。

また第五章では，前二章において論じたアーキテクチャの改良や最適化に関して，実際の評価データに基いてそれらの有効性を明らかにする。また，計算機アーキテクチャにおいて特に重要なポイントであるメモリへのアクセスについて，その特性や実際のハードウェア構成との関連性を評価する。

最後に第六章では，本研究の結論を述べるとともに，今後の課題について概観する。

第2章
論理型言語とその処理方式

特定のプログラミング言語（または言語群）に向いたプロセッサのアーキテクチャを論ずる上で，対象となる言語の持つ性質や特徴を知ることは極めて重要である。特に，専用プロセッサの最大の特質である高速処理の実現のためには，プログラムがどのように処理され，またどのような操作が頻繁に成されるのかについて，深い理解が必要である。

そこで本章では，論理型言語の仕様や特徴，更には具体的な処理の方法について，代表的な論理型言語である Prolog を対象に論ずる。Prolog は 1971 年に Alan Colmerauer により考案され，その後 Robert Kowalski によってプログラミング言語として位置付けられた論理型言語である [Furukawa 84, Kowalski 74]。Prolog の最大の特徴は，一階述語論理に基くホーン節によってプログラムが表現されること，即ち「論理」に基礎を置いていることである。従って，プログラムの数学的な取扱い，例えばプログラムの検証，合成，変換などの問題に対する相性がよく，これらの分野において Prolog を対象とした研究が数多く成されている。また，プログラムの実行過程が AND-OR Tree の探索に基いていることから，定理証明やゲーム木の探索などの人工知能の分野での問題にも適している。

一方別の観点からは，Prolog を強力な記号処理言語と捉えることもできる。Prolog はこの分野での先駆的な存在である LISP と同様に，リスト構造やより複雑な構造体を取扱う機能を持っている。その上，ユニフィケーションという強力な操作によって，不完全データ構造 (Incomplete Data Structure) の利用や，複雑な構造体間のパターン・マッチングが可能であるため，記号処理の能力は LISP を上回るものであると言える。また，構造体の分解や合成が極めて自然な形で記述できることも，LISP に優る特質である。

さて，プログラミング言語として幅広く用いられるためには，記述能力の高さだけではなく，処理系の質，即ち高い処理速度と記憶空間の効率的な使用が求められる。この点において，初期の Prolog 処理系は全く不充分なものであり，実用的な言語であるとはみなされていなかった。これを大幅に改善したのが，1977 年に David H. D. Warren らによって開発された DEC-10 Prolog の処理系である [Warren 77, Bowen 81]。Warren らは，それまでのインタプリタによる実現に変えて，初めてコンパイラを導入するとともに，変数の領域割当や Tail Recursion に関する基本的な最適化を行うことにより，処理速度と記憶効率の両面で 10 倍以上と言う飛躍的な性能向上を達成した。その結果，Prolog の価値の見直しとユー

ザの急速な拡大をもたらすとともに，筆者らを含む数多くの研究者によって，専用マシンや処理系に関する研究が成される契機ともなったのである。

以下本章では，まずPrologのプログラミング言語としての構造を概説した後，その二大特徴とも言うべきユニフィケーションとバックトラックの処理の方式について述べ，更に基本的な最適化手法について議論する。また，筆者らが開発した逐次型推論マシンの基本言語であるESP，その土台となっている言語KL0，更に並列推論マシンの基本言語であるKL1について，それぞれの特徴を簡単に述べる。

2.1 言語の構造

Prologのプログラムは，事実(fact)，規則(rule)，及び質問(query)の三つの要素から成り立っていると見ることができる。例えば；

```
father(家康，信康).
father(家康，秀忠).
father(家康，秀康).
```

は全て「事実」であり，「家康は信康の父である」などを表現している（と解釈することができる）。

また；

```
brother(X, Y):- father(F, X), father(F, Y).
```

は「規則」であって，「FがXの父であり，かつFがYの父でもあるならば，XとYは兄弟である」という論理的な推論規則を示している。ここでX, Y, Fは変数であって，この「規則」に関する限りその値はどのようなものであっても良い。

なお，「事実」と「規則」は総称してクローズ（または節）と呼ばれる。「規則」の左辺（':-'の左側）をヘッド，右辺をボディと言う。従って，「事実」はボディのない特別なクローズであると考えることができ，ユニット・クローズと呼ばれることもある。また，ボディを構成する要素（上の例での`father(F, X)`及び`father(F, Y)`）はゴールと呼ばれ，これらはANDの関係にある。更に，名前と引数の数が同じヘッドを持つクローズの集合は述語と呼ばれ，一つの述語に含まれる事実や規則はORの関係にある。

さて，これらの事実と規則を前提とした時，「信康と秀忠は兄弟であるか」という命題は，以下の三段論法によって真であることが証明できる。

1. 家康は信康の父であり，かつ家康は秀忠の父である。
2. FがXの父であり，かつFがYの父でもあるならば，XとYは兄弟である。
3. 故に信康と秀忠は兄弟である。

一方，Prologの処理系に対して；

```
:- brother(信康，秀忠).
```

という「質問」*を行うと，前述の三段論法を逆方向にたどる形の後向き推論によって，図2-1に示すような実行過程を経て質問が真であることが導き出される。即ち；

*ヘッドのない特別なクローズと考えることができる

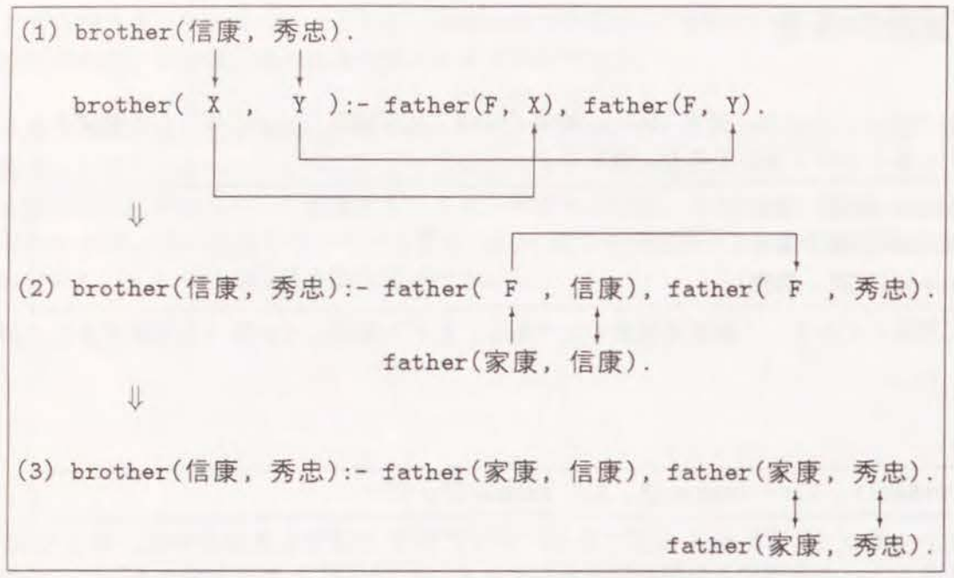


図 2-1: brother(信康, 秀忠) の実行

(1) 「質問」；

```
:- brother(信康, 秀忠).                    (a)
```

のゴール brother(信康, 秀忠) とユニファイ可能なヘッドを持つクローズを探す。クローズ；

```
brother(X, Y):- father(F, X), father(F, Y).    (b)
```

が見つかり，brother(信康, 秀忠) と brother(X, Y) をユニファイする。その結果，X = 信康, Y = 秀忠となる。

(2) (b) の最初のゴール father(F, 信康) (X = 信康となっている) とユニファイ可能なヘッドを持つクローズを探す。クローズ；

```
father(家康, 信康).                        (c)
```

が見つかり，father(F, 信康) と father(家康, 信康) をユニファイする。その結果，F = 家康となる。(c) はゴールを持っていないので (b) に戻る。

(3) (b) の二番目のゴール father(家康, 秀忠) (F = 家康, Y = 秀忠となっている) とユニファイ可能なヘッドを持つクローズを探す。クローズ；

```
father(家康, 秀忠).                        (d)
```

が見つかり，father(家康, 秀忠) と father(家康, 秀忠) をユニファイする。(d) はゴールを持っていないので (b) に戻り，(b) にもゴールがないので (a) に戻り，(a) にもゴールがないので終了する。

以上のような実行過程を，手続型言語のそれと類比すると；

クローズ　　≈ サブルーチン
ゴール　　≈ サブルーチン呼出
ユニフィケーション ≈ 引数の受渡し

のように考えることができよう。但しユニフィケーションは，手続型言語の単純な引数受渡しとは異なり，以下のように双方向の代入と比較を兼ね備えたものである。

- ゴールからヘッドへの代入　brother(信康, 秀忠) → brother(X , Y)
- ヘッドからゴールへの代入　father(F , 信康) ← father(家康, 信康)
- ゴールとヘッドの比較　father(家康, 秀忠) ↕ father(家康, 秀忠)

さて，以上の議論に基くと，Prolog は「ユニフィケーションと言う特殊な引数受渡し機構を持った，サブルーチン呼出しだけからなる言語」と言うことになる。しかし，Prolog はバックトラックというもう一つの重要な機構を持っており，これが言語の大きな特徴となっている。

バックトラックの機構は，AND-OR 木の左優先／深さ優先の探索のための機構とみなすことができる。即ち，クローズはそのゴールを分枝とする AND ノードであり，また特定のゴールとユニファイ可能なヘッドを持つクローズの集合が OR ノードとなる。探索は，ゴールとヘッドのユニフィケーションにより行なわれ，ユニフィケーションが失敗すると最後に通った OR ノードへの後戻りが発生する。これがバックトラックである。

例えば，前述のプログラムに以下の事実／規則を追加することを考える。

```
elder(信康, 秀忠).
elder(信康 , 秀康).
elder(秀忠 , 秀康).

elder_brother(X, Y):- father(F, X), father(F, Y), elder(X, Y).
```

このプログラムに対して；

```
:- elder_brother(秀忠, B).
```

即ち，「秀忠の弟は誰か」という質問を行うと，それが秀康であることが図 2-2に示すような手順で求められる。また，これに対応する AND-OR 木と，その探索順序は図 2-3に示すものとなる。

```
(1) elder_brother(秀忠, B).
                  ↓   ↓
    elder_brother( X  , Y):- father(F, X), father(F, Y), elder(X, Y).
      ⇓

(2) elder_brother(秀忠, B):- father( F  , 秀忠), father(F, B), elder(秀忠, B).
                              ↑     ↕
                           father(家康, 秀忠).
      ⇓

(3) elder_brother(秀忠, B):- father(家康, 秀忠), father(家康,  B  ), elder(秀忠, B).
                                                 ↕     ↑
                                              father(家康, 信康).
      ⇓

(4) elder_brother(秀忠, 信康):- father(家康, 秀忠), father(家康, 信康), elder(秀忠, 信康).
                                                                      ↓
                                                                  elder(信康, 秀忠).
                                                                  elder(信康, 秀康).
                                                                  elder(秀忠, 秀康).
      ⇓ (backtrack)

(5) elder_brother(秀忠, B):- father(家康, 秀忠), father(家康,  B  ), elder(秀忠, B).
                                                 ↕     ↑
                                              father(家康, 秀忠).
      ⇓

(6) elder_brother(秀忠, 秀忠):- father(家康, 秀忠), father(家康, 秀忠), elder(秀忠, 秀忠).
                                                                      ↓
                                                                  elder(信康, 秀忠).
                                                                  elder(信康, 秀康).
                                                                  elder(秀忠, 秀康).
      ⇓ (backtrack)

(7) elder_brother(秀忠, B):- father(家康, 秀忠), father(家康,  B  ), elder(秀忠, B).
                                                 ↕     ↑
                                              father(家康, 秀康).
      ⇓

(8) elder_brother(秀忠, 秀康):- father(家康, 秀忠), father(家康, 秀康), elder(秀忠, 秀康).
                                                                      ↕     ↕
                                                                  elder(秀忠, 秀康).
```

図 2-2: elder_brother(秀康, B) の実行

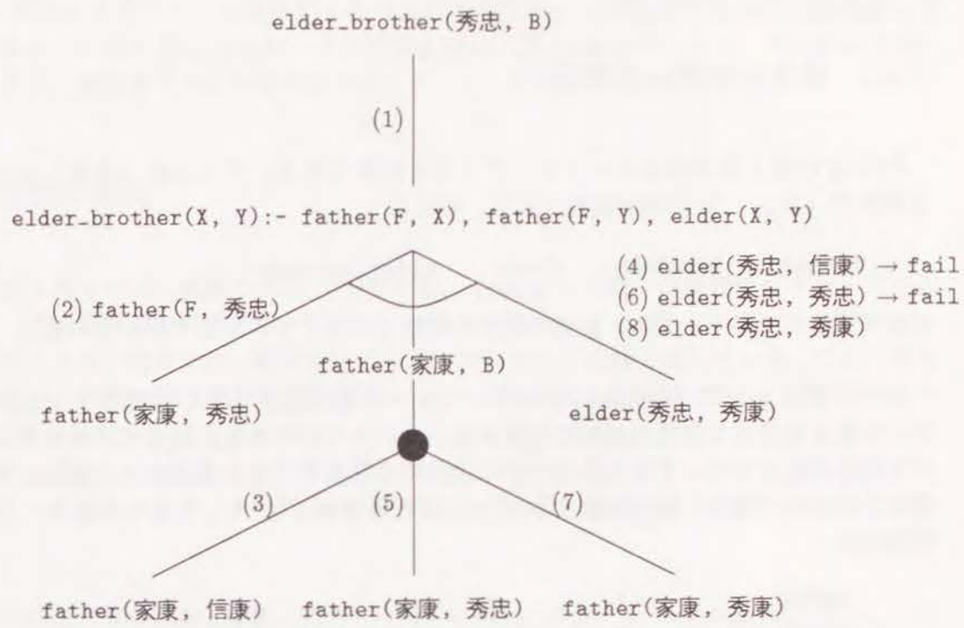


図 2-3: elder_brother(秀忠, B) の AND-OR 木

なお，ここで注意しなければならないのは，バックトラックが単に実行順序の変更を行うだけではなく，実行環境の復元も含んでいることである。例えば elder_brother(秀康, B) の例では，バックトラックによって (4) から (5) へ移行する際に，変数 B の状態が元に戻されている。即ち，(3) における変数 B と信康のユニフィケーションを行う以前の状態に復元されているのである。このユニフィケーションの無効化は Undo と呼ばれ，ユニフィケーションとバックトラックの双方に関係する重要な操作である。

以上述べてきたように，Prolog は；

- ユニフィケーションという強力な引数受渡し機構と，
- バックトラックという特殊な制御機構を持った，
- サブルーチン呼出しだけからなる言語

ということができる。従って，ユニフィケーションとバックトラックという，二つの強力でかつ複雑な機構をいかに実現するかが，Prolog 処理の最も重要なポイントであるといえる。

2.2 ユニフィケーション

2.2.1 基本的なデータ表現

Prologの最も基本的なデータは，アトムと整数である。アトムは（通常）小文字で始まる英数字（及び'_'）の列で表現される。例えば；

```
father    brother    elder    elder_brother
```

は全てアトムである。また，前述の例での家康などもアトムとして用いている*。

さて，アトムとアトムのユニフィケーションの規則は，「同じ名前のアトムのユニフィケーションは成功し，それ以外は失敗する」というものである。従って，ユニフィケーションを行う側にとって，アトムの具体的な名前である文字列は意味がなく，個々のアトムを識別するための「番号」付けが成されていれば充分である†。そこで多くの場合，アトムの内部表現は；

```
father        → 1
brother       → 2
elder         → 3
elder_brother → 4
家康          → 5
```

のように，適当な整数が用いられる‡。

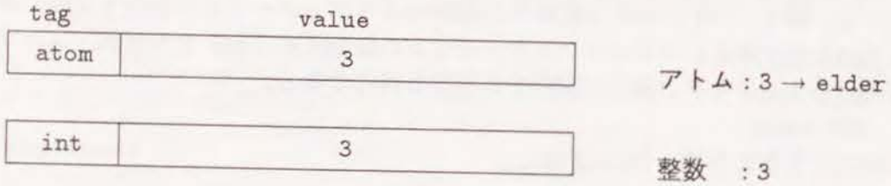


図 2-4: アトムと整数の内部表現

*一般の処理系が漢字列をアトムとして取扱うという訳ではない。

†もちろん「番号」から「名前」を求める手段（及びその逆）は必要であるが，ユニフィケーションとは直接関係しない。

‡「名前」を記憶している領域のアドレスを用いると「番号」から「名前」への変換が容易になるが，ガベージ・コレクションなどによって領域が移動すると「番号」の付け替えが必要となり，かえって不利であることが多い。

一方，整数の自然な内部表現はもちろん整数自身であるが，例えば「整数'3'」と「番号'3'が割り振られたアトム」とは区別しなければならない。この区別のために用いられるのがタグである。一般にPrologのデータの内部表現は，図2-4に示すように，データの「型」を示すタグと，数値やアトムの番号などを示す「値」の部分とに分かれている。

2.2.2 変数の表現

次に問題となるのは，変数の表現方法である。Prologの変数の有効範囲はクローズの内部であり，手続型言語の局所変数と同様にローカルなスコープを持っている。また，クローズが「呼出された」時点では，変数は「何も入っていない」状態となっている。この「何も入っていない」状態，即ち未定義状態がPrologの変数の大きな特徴であり，手続型言語やLISPの変数の初期状態である「何が入っているか分からない」*状態とは大きく異なっている。即ち，Prologの変数は未定義状態の場合にのみ，任意の値をユニフィケーションによって代入することができ，一旦代入が行われると別の値に変更することはできないのである。

例えば，前述の`brother(信康，秀忠)`の例において，クローズ`brother(X, Y):-...`の状態は以下のように変化した。

(1) `brother(X, Y):- father(F, X), father(F, Y).`
(2) `brother(信康, 秀忠):- father(F, 信康), father(F, 秀忠).`
(3) `brother(信康, 秀忠):- father(家康, 信康), father(家康, 秀忠).`

この時，変数XとYはbrotherのヘッド・ユニフィケーションによって，未定義状態から値信康と秀忠を持っている状態に変化する。従って，後のユニフィケーションにおいては，例えばXに信康以外の値を代入することはできず，アトム信康以外のものとのユニフィケーションは失敗するのである†。同様にFは最初のゴールに関するユニフィケーションによって，未定義状態から家康を値とする状態に変化し，二番目のゴールのユニフィケーションではアトム家康が対象となる。

以上の議論によって，Prologの変数を表現するためには，未定義状態であることを示すなんらかの方法が必要であることが判る。そこで，図2-5に示すように，タグがundefであるようなデータが未定義状態の変数であるとすることが多い‡。そして，ユニフィケーションの際に一方のタグがundefで，他方のタグが例えばatomであれば，undefである未定義変数にタグと値の双方が代入される。なお，未定義変数の「値」の部分は何であっても構

*Cでは'0'が初期状態であると規程されている。

†もちろん他の未定義状態の変数とのユニフィケーションは成功し，信康が代入される。

‡後述するように，別の有力な方法もある。

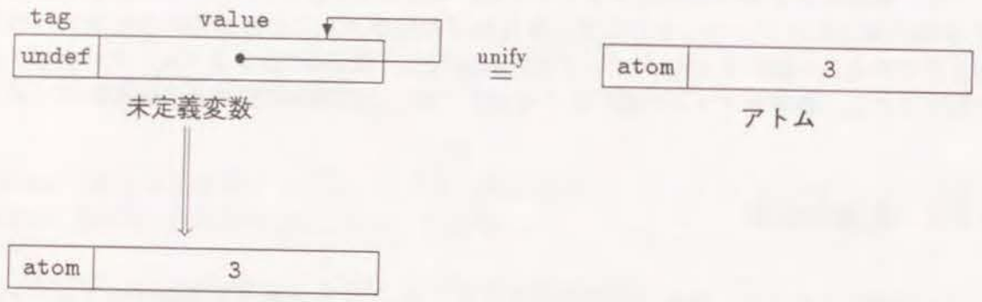


図 2-5: 未定義変数の内部表現

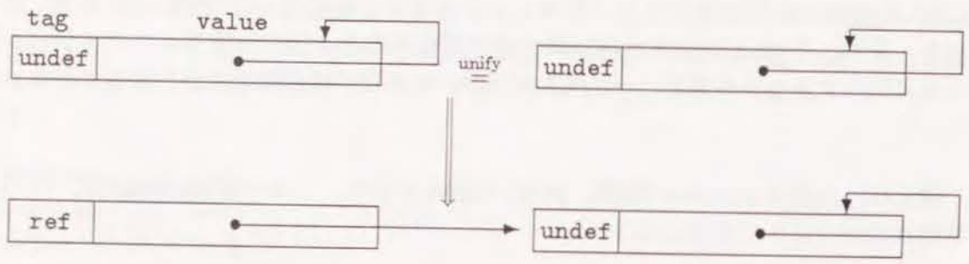


図 2-6: 未定義変数どうしのユニフィケーション

わないが，他の処理との関係で自分自身を指すポインタを入れておくと便利であることが多い。

さて，一方が未定義変数である時のユニフィケーションでは代入が行われると述べたが，双方が未定義変数の場合には単純な代入を行う訳にはいかない。この場合には，二つの変数を「同じもの」にする処理が必要となる。例えば，前述の `elder_brother(秀忠, B)` の例では，クローズ `elder_brother(X, Y):-...` の状態は以下のように変化した。

```
(1) elder_brother(X, Y):- father(F, X), father(F, Y), elder(X, Y).
(2) elder_brother(秀忠, B):- father(F, 秀忠), father(F, B),
                              elder(秀忠, B).
(3) elder_brother(秀忠, B):- father(家康, 秀忠), father(家康, B),
                              elder(秀忠, B).
    ⋮
(4) elder_brother(秀忠, 秀康):- father(家康, 秀忠), father(家康, 秀康),
                                elder(秀忠, 秀康).
```


即ち，`elder_brother(秀忠, B)` と `elder_brother(X, Y)` とのユニフィケーションによりBとYは同じものになり，(4)におけるユニフィケーションによってアトム秀康がYを経由してBに代入される。

この「同じものにする」処理は，Reference Pointerという特殊なポインタを用いて行う。即ち，図2-6に示すように，未定義変数どうしのユニフィケーションでは，一方の変数に他方を指すポインタを代入し，タグにReference Pointerであることを示す`ref`をセットする。一方，Reference Pointerに対するユニフィケーションは，そのポインタが指しているデータに対して行い，ポインタとそれが指すデータが「同じもの」であることを実現する。

従ってユニフィケーションの際には，Reference Pointerであるか否かを判断し，そうであればそれが指しているデータを読む，という操作が必要となる。この操作はデレファレンスと呼ばれ，Prolog処理において極めて重要な操作の一つである。なお，Reference Pointerが指している未定義変数が，他の未定義変数とのユニフィケーションによってReference Pointerとなることがあるため，一般には任意の長さのReference Pointerの連鎖が生ずることがある*。

なお，未定義変数を表現する別の方法として，自身を指すReference Pointerを用いる方法がある[Warren 83]。この方法は，タグの種類を減らす効果の他，未定義変数の「移動」が容易に行えるという利点がある。即ち，未定義変数を別の場所にコピーすると，自動的に元の場所を指すReference Pointerが得られる。従って，あるデータをコピーする際に，そのデータのタグに依存せず単純なコピーを行うことができる†。一方，未定義であるか否かの判定に，タグ判定の他にポインタの値の判定が加わるため，ユニフィケーション時の処理が重くなるという欠点がある。従って，`ref`タグを用いるほうが有利であるという報告もあるが[Touati 87]，ハードウェア・サポートの有無やそのコストなどを勘案すると，どちらが有利かの判断を下すことは難しい。

さて，変数を取扱うに当っての重要な項目の一つに，変数をどこに置いておくか，即ち変数の割付の問題がある。前述のようにPrologの変数は一つのクローズに「ローカルな」ものであるので，手続型言語の局所変数と同様にスタックの上に割付けるのが自然である。即ち，変数といくつかの制御情報からなるスタック・フレームを，クローズが呼び出された時に生成し，実行完了時にスタックから除去する方法が考えられる。

このフレームは「環境」(Environment)と呼ばれ，具体的には図2-7に示すように；

- 一つ上のEnvironmentへのポインタ (**old-E**)

*これを避ける効率的な方法はない。

†undefタグを用いる場合は，undefならばrefを作り，それ以外の場合はコピーする，という処理となる。

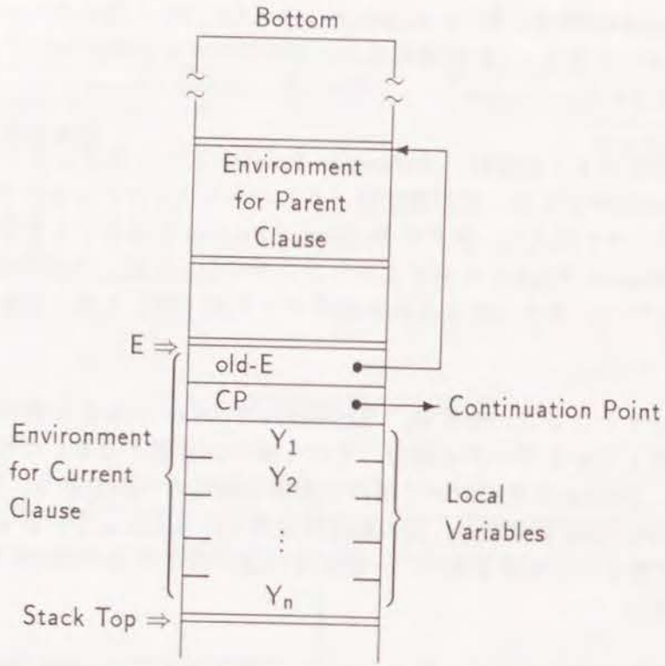


図 2-7: Environment と局所変数

- 復帰アドレス (CP)
- 局所変数 (Y_n)

から構成される。また，実行中のクローズの Environment のベース・アドレスは頻繁に使用されるため，レジスタ (E) に保持しておくのが普通である。

このような方法を用いた場合，未定義変数どうしのユニフィケーションに，一定の制限が加わることに注意しなければならない。即ち，未定義変数 X と Y のユニフィケーションにおいて，X が Y よりもスタックのボトム側にある場合，Y に X への Reference Pointer を代入しなければならない。これを逆にして X に Y への Reference Pointer を代入すると，Y を含む Environment がスタックから除去された場合に，X の内容が「存在しないセルへのポインタ」となってしまうからである。

なお，Prolog の変数はスコープの観点からは局所的であるが，全てを局所変数として取扱うことはできない。即ち，Environment に割付けることができない「大域変数」が存在するが，これに関しては後述する。また，バックトラックとの関係で，変数への代入を無効化するための処置も必要であるが，これに関しても後に述べる。

2.2.3 基本的なユニフィケーション

以上の議論をまとめると，アトム，整数，及び未定義変数（局所変数）に関する，二つのデータ X と Y のユニフィケーション操作は以下のようになる。

(1) X をデレファレンスしその結果を X' とする。
(2) Y をデレファレンスしその結果を Y' とする。
(3) 下表に従った操作を行う。

tag(X')	tag(Y')		
	undef	atom	int
undef	$Y' \leftarrow$ ref to X' if $X' < Y'$ $X' \leftarrow$ ref to Y' if $X' > Y'$	$X' \leftarrow Y'$	$X' \leftarrow Y'$
atom	$Y' \leftarrow X'$	success if $X' = Y'$ fail otherwise	fail
int	$Y' \leftarrow X'$	fail	success if $X' = Y'$ fail otherwise

但し，$X' < Y'$ は X' が Y' よりもスタックのボトム側にあることを意味する。

しかしこれは，ユニフィケーションの「一般的な」操作であり，実際はより簡単な操作で済む場合が多い。即ち，ゴールの引数とヘッドの引数のユニフィケーションにおいて，ヘッドの引数は実行前に（コンパイル時に）何であるかが判っているため，この知識を用いて操作を簡略化することができるのである。

以下，ユニフィケーションの簡略化について議論するが，その前にゴールの引数をどこに置いておくかを明らかにする必要がある。これにはいくつかの方法があり，例えば [Warren 77] では，復帰アドレスの直後にゴール引数が何であるかを示す情報（例えば定数の値，呼び出し元の Environment における局所変数の位置など）を置いている。また，いくつかの LISP マシンで見られるように，引数を評価した値を呼出しの前にスタックへプッシュしておく方法も考えられる。更にこれを発展させた方式として，後述する PSI-I や [Warren 83] のように，引数の評価値をレジスタにセットしておく方法もある。

これらの方式には一長一短があるが，以下の議論においては現在最も効率的と考えられている [Warren 83] の方法を前提とする*。この方法では，n 個の引数を格納するための引数レジスタ $A_1, A_2, \ldots, A_n$ が用いられ，呼出しの前にゴール引数の評価値がこれらのレジスタにセットされる。例えば前述の `brother(信康, 秀忠)` の例における，クローズ `brother(X, Y):-...` の最初のゴール `father(F, X)` では；

*効率的である理由については後述する。

A_1 未定義の局所変数 F への Reference Pointer
A_2 局所変数 X の値信康

がそれぞれセットされる。

さて，ユニフィケーションの簡略化であるが，最も簡単であるのはヘッド引数が未定義変数であると判っている場合である。Prolog の変数は初期的には未定義状態であるので，ヘッドに出現する「初出の」変数に関するユニフィケーションがこれに相当する。この場合，前述のユニフィケーション規則に従えば，この変数を Y とし，対応するゴール引数を A とした時；

(1) A のデレファレンス結果を A' とする。
(2) Y を未定義状態とする。
(3) A' が未定義変数であり；
  (a) $A' < Y$ であれば Y に A' への Reference Pointer を代入する。
  (b) $A' > Y$ であれば A' に Y への Reference Pointer を代入する。
(4) A' がアトムか整数であれば，その値を Y に代入する。

となるが，この操作は更に簡略化することができる。

まず，A' が未定義変数の場合，A' は Y が含まれている Environment よりもスタックのボトム側にある Environment に含まれているため，(3)(b) は起こりえない。従って，いずれの場合も Y への代入が行われるため，(2) における Y の初期化も不要である。また，(1) のデレファレンスを行わずに A を Y に直接代入することもできる。この場合，A がアトムか整数であるか，または未定義変数を直接指している Reference Pointer であればもちろん問題ない。それ以外の場合でも，代入後の Y は A' を直接または間接に指す Reference Pointer となり，A' と等価になるので論理的には正しい。更に，この操作では A' へ至る Reference Pointer の段数を増やしていないため，同様のユニフィケーションを繰返し行うことによる段数の増加を心配する必要もない。従って上記の操作は，「A を Y に代入する」という極めて単純な操作とすることができる。

次に簡略化できるユニフィケーションは，ヘッド引数がアトム（または整数）であると判っている場合である。この場合，アトムを c, ゴール引数を A とした時；

(1) A のデレファレンス結果を A' とする。
(2) A' が未定義変数であれば A' に c を代入する。
(3) A' がアトムであれば A' と c を比較し；
  (a) 等しければ成功
  (b) 等しくなければ失敗
(4) A' が整数であれば失敗。

となる。この操作の核心的な部分は A' のタグの判定であり，タグが undef, atom, int のいずれであるかにより三方向の分岐が行われる。また，デレファレンスにも ref か否かのタグ判定が伴うため，これを加えると四方向の分岐となる*。このようにタグによる処理の分岐が多方向であることが Prolog 処理の特徴の一つであり，LISP のような特定のデータ型であるか否かという二方向分岐中心のものとはかなり異なっている。

最後のタイプのユニフィケーションは，「一般的な」ユニフィケーションである。これは；

```
p(X, X):- ...
```

の第二引数のユニフィケーションのように，双方がどのようなデータか実行前には判らない場合に行われる。この場合には前述の一般的なユニフィケーション規則が適用され，より複雑な多方向分岐が行われる。

以上をまとめると，ユニフィケーションは一方のデータ（ヘッド引数）が；

i) 未定義変数であることが実行前に判明している；
ii) アトムまたは整数であることが実行前に判明している；
iii) どのようなデータであるかが実行前には不明；

の三つの場合に分けられる。操作は上記の順序で複雑になり，ii), iii) ではタグによる多方向の分岐が処理の中核となる。

2.2.4 構造体のユニフィケーション

前項までの議論は，アトム及び整数と言う構造を持たない「アトミックな」データと，局所変数についてのユニフィケーションに関するものであった。本項では，Prolog の重要な特徴の一つである構造体（または複合項）に関するユニフィケーションと，それに関連する大域変数について議論する。

Prolog では構造体は，f(a, X) のように，「関数」の形で記述される。これは，a と X の二つの要素からなる f という名前の構造体と考えることもできるし，また f, a, X の三要素のベクタと解釈することもできる。構造体の要素の数は任意であり，また；

```
f(a, g(X, Y, b))
```

のように要素に構造体を持つこともできる。

*一般的には，ref, undef, atom, 及び「それ以外」の四方向となる。

なお，Prologでは構造体に関するいくつかの便利な記法がある。例えばリスト・セルについては'.'(X, Y)が基本的な記法であるが，[X|Y]と記述することもできる†。更に[X|[Y|[Z|[]]]]（[]はnil）を[X,Y,Z]と記述することもできる。また，'+'(X, Y)をX + Yと記述する演算子記法も許されている。例えば；

```
brother(X, Y):- father(F, X), father(F, Y).
```

も演算子記法の一種であり；

```
':-'(brother(X, Y), ','(father(F, X), father(F, Y)))
```

と解釈される。

さて，構造体Sと任意のデータXとのユニフィケーションの規則は，以下のように定められる。

(1) Xが未定義変数であれば，XにSを代入して成功。

(2) Xが同じ構造を持つ構造体（即ち名前と要素数が同じ）であれば，SとXの各要素についてユニフィケーションを行う。

(3) それ以外は失敗。

この規則の中の(2)が持つ再帰性が，構造体のユニフィケーションの重要な特徴となっている。

構造体のユニフィケーションにおいても，アトミックなデータと同様に，操作の簡略化を行うことができるが，構造体の要素として現れる変数の取扱いや，構造体の「生成」に関していくつかの注意が必要である。

まず，ヘッド引数が未定義変数であると判っている場合，即ち初出の変数に関するユニフィケーションは，ゴール引数が構造体であっても単純な代入とすることができる。但しこの場合，ゴール引数を格納する引数レジスタなどには，構造体を適切に表現する1ワードのデータが格納されていなければならない（具体的な表現については後述）。

次に，ヘッド引数として陽に構造体が与えられ，かつゴール引数が未定義変数の場合について考える。この場合，未定義変数に対して構造体を代入する必要があるが，1ワードで表現可能なアトミックなデータとは違って，単純に代入を行う訳にはいかない。即ち，複数ワードからなる構造体を「生成」し，かつ生成した構造体を適切に表現する1ワードのデータを未定義変数に代入しなければならない。この構造体の生成方法が，Prologの処理方式の中で重要なポイントの一つとなっている。

†実現方式の都合で[X|Y]のみをリスト・セルと解釈するものも多い（例えばESP）。

さて，構造体の生成方法を論ずる前に，構造体の要素として現れる変数の取扱いを考える必要がある。例えばクローズ；

```
p(X, f(a, Y)):- ...
```

がゴールp(A, B)によって呼び出される場合を考える（ゴール引数Bは呼出しの時点で未定義であるとする）。前述のように，Xは局所変数としてクローズのEnvironmentに割付けられ，pの実行が完了した時点でスタックから除去される。これは，ゴール引数Aがどのような値であっても，呼出し側からXへの参照パスが存在しないことに基いている。

一方，Yについては事情が異なっている。即ち，Bにf(a, Y)（を表現するデータ）が代入されるため，呼出し側からYに対する間接的な参照パスが存在する。従って，pの実行が完了してもYの変数セルを除去してしまうことができないため，YをEnvironmentに割付けてはならない。

このYのようにEnvironmentに割付けることができない変数は「大域変数」と呼ばれ，Environmentを保持する「ローカル・スタック」とは別のスタックである「グローバル・スタック」に割付けられる。グローバル・スタックはクローズの実行が終了しても縮まないが，後述するようにバックトラックによって縮めることができるため，「スタック」の名称が与えられている。

さて，構造体の生成方法であるが，これには構造体複写法(Structure Copying)と構造体共有法(Structure Sharing)の二つの方法がある。

複写法では，構造体の全体がグローバル・スタック上に生成される。即ち，先に述べたp(X, f(a, Y))の例では，図2-8に示すように，グローバル・スタック上に（例えば）要素数3，アトムfとa，及びYの変数セルからなるデータ構造が生成される。また，呼出し側の変数Bにはこのデータ構造の先頭へのポインタに，構造体を指示するポインタであることを示すタグstrctを付したデータが代入される。なお，Yを局所的に参照するために，p(X, f(a, Y))のクローズのEnvironmentにも変数セルが取られ，グローバル・スタック上のYへのReference Pointerが格納される。なお，生成した構造体のn番目の要素へのアクセスは，構造体の先頭へのポインタにnを加えたものをアドレスとして，簡単に行うことができる。

一方，共有法では構造体を，図2-9に示すように，固定的な部分と可変の部分に分けて表現する。例えばf(a, Y)では，要素数3，及びアトムfとaは固定的な部分であり，Yが可変部分である。固定的な部分はスケルトン(Skeleton)と呼ばれ，何回構造体を生成しても常に同じものであるため，予めコード領域などに置かれる。可変部分，即ち大域変数は，構造体を生成するたびに異なる（可能性がある）ので，グローバル・スタック上に置かれる。

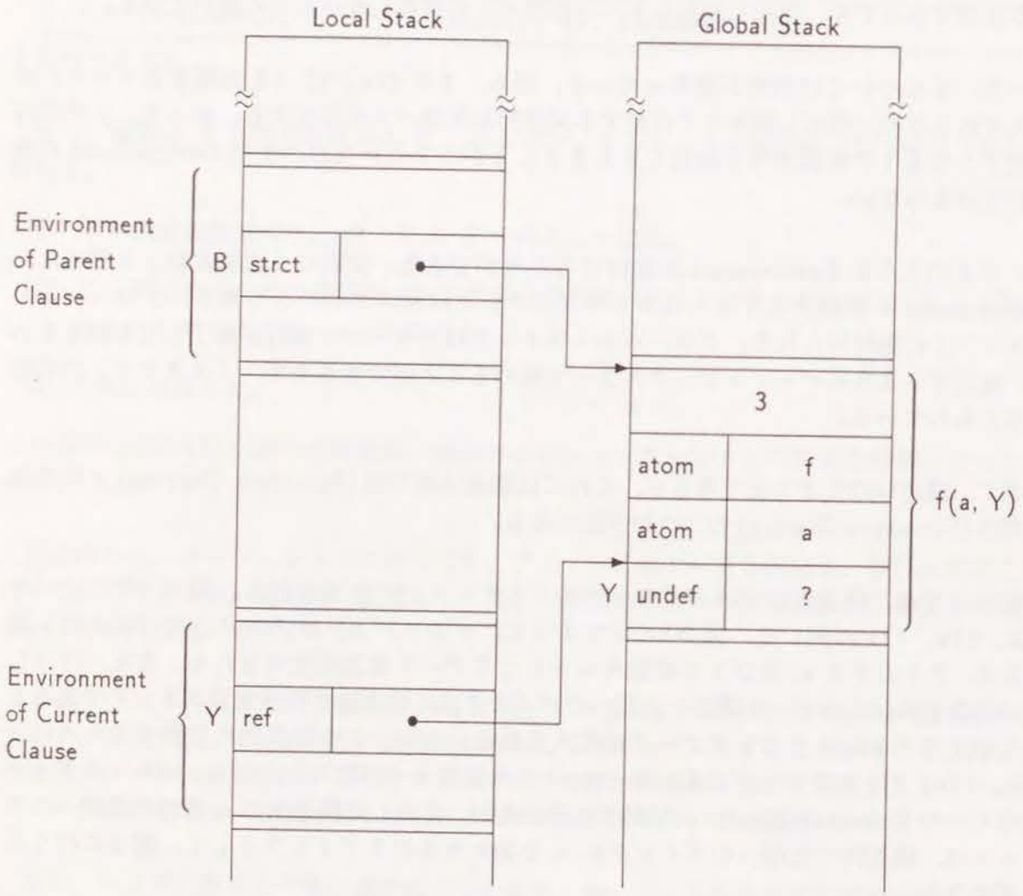


図 2-8: 構造体複写法

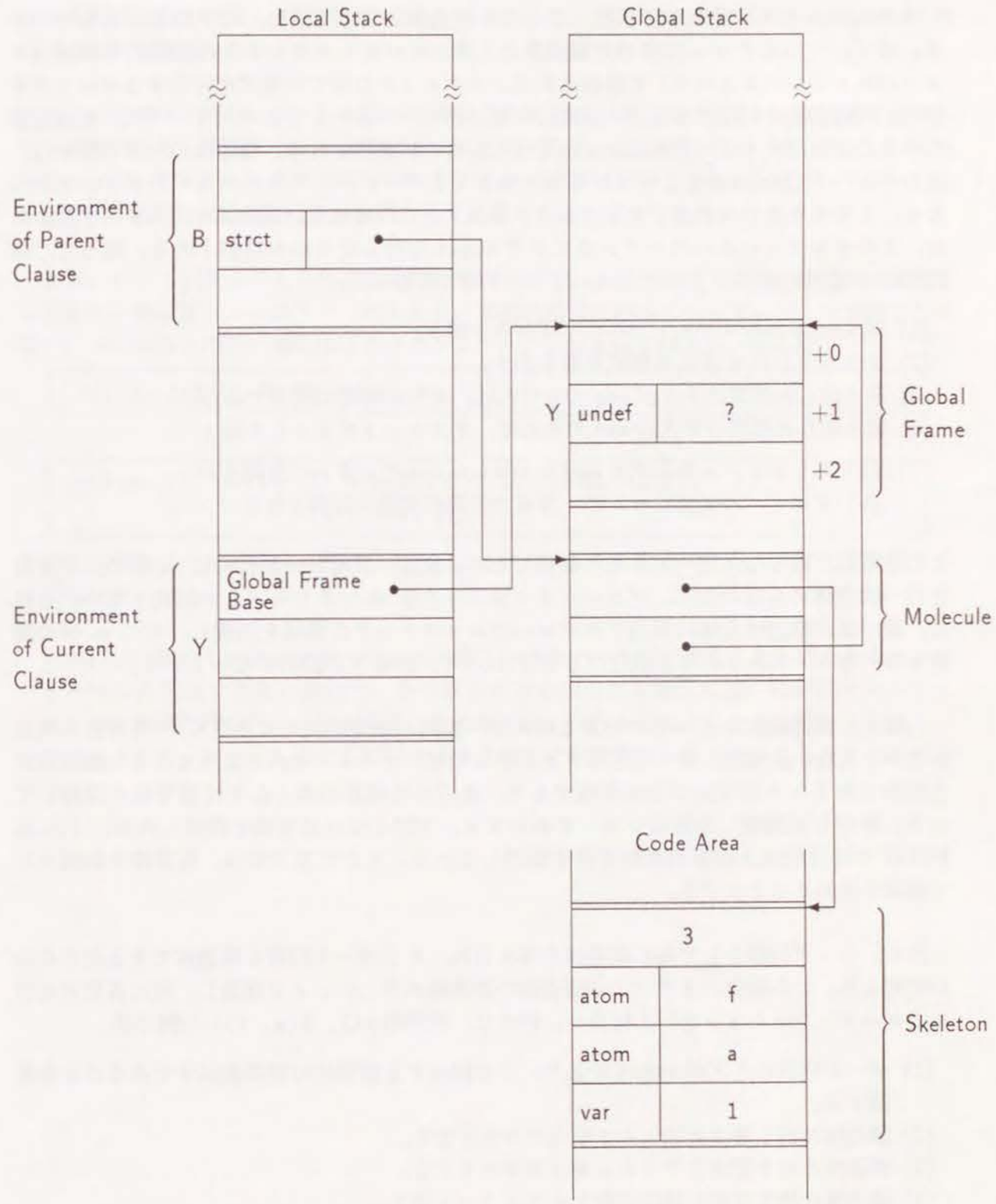


図 2-9: 構造体共有法

スケルトンと大域変数を結び付けて一つの構造体とする方法は，以下のようなものである。まず，一つのクローズ中の大域変数は，グローバル・スタック上の連続的な領域（グローバル・フレームという）に確保される。スケルトンは全ての要素に対応するエントリを持ち，大域変数に対応するエントリにはグローバル・フレーム中のオフセットに，大域変数であることを示すタグ（例えば var）を付したものが格納される。構造体の生成の際には，モレキュール (Molecule) と呼ばれるスケルトンとグローバル・フレームへのポインタからなる，2ワードのデータがグローバル・スタック上に作られる。構造体を代入すべき変数には，このモレキュールへのポインタにタグ strct を付したものが格納される。従って，構造体の n 番目の要素のアクセスは，以下の手順で行われる。

(1) Molecule からスケルトンへのポインタを得る。
(2) スケルトンの n 番目の要素を取り出す。
(3) 取り出した要素のタグが var でなければ，それが実際の要素である。
(4) 取り出した要素のタグが var であれば，オフセットを i とした時；
    (a) モレキュールからグローバル・フレームへのポインタを得る。
    (b) グローバル・フレームの i 番目の要素が実際の要素となる。

この手順は，複写法に比べてかなり複雑である。反面，共有法の場合には m 個の大域変数を持つ構造体の生成の際に，グローバル・スタックを $m+2$ ワードしか消費しない。これは，複写法が構造体全体に相当するグローバル・スタックの領域を消費し，かつ m 個の変数をローカル・スタックにも割付ける*のに比べて，かなりの節約になっている。

しかし，構造体が小さいものである時には，節約の効果が極めて小さく†，共有法の利点が生かされない。実際，多くの応用プログラムでは，リスト・セルのような小さい構造体が支配的であるため複写法の方が有利であり，最近の処理系のほとんどは複写法を採用している。筆者らが開発した推論マシンにおいても，PSI-I では共有法を採用したが，PSI-II, PSI-III では性能向上のために複写法を採用している。そこで以下では，複写法を前提として議論を進めることにする。

次に，ヘッド引数として陽に構造体が与えられ，かつゴール引数も構造体である場合について考える。この場合，まず二つの構造体の要素数が等しいことを確認し，次に各要素についてユニフィケーションを行えば良い。例えば，前述の `p(X, f(a, Y))` の例では；

(1) ゴール引数のタグが strct であり，その指示する構造体の要素数が3であることを確認する。
(2) 構造体の第1要素とアトム f をユニファイする。
(3) 構造体の第2要素とアトム a をユニファイする。
(4) 構造体の第3要素を局所変数 Y をユニファイする。

*最適化により割付けないことも多い。後述。
†Molecule のためにかえって消費量が大きくなることもある。

という処理が行われる。この処理の中の (2) ～ (4) は，前項で述べたアトミックなデータや局所変数に関するユニフィケーションと，ほとんど同様である。

ここで注意すべきは，ゴール引数が未定義変数であるか構造体であるかが実行前には判らないことである。即ち，(1) の処理において前項と同様，undef, strct, デレファレンスのための ref，及びそれら以外の四方向のタグ分岐が必要であるとともに，(2) ～ (4) の処理もゴール引数が undef か strct かによって処理の内容を変更しなければならない。

なお，ヘッド引数に与えられた構造体がネストしている場合には，内側の構造体に対応する要素を一時変数とユニファイ（代入）し，外側の構造体のユニフィケーションが終了した後で，一時変数と内側の構造体とのユニフィケーションを行えば良い。例えば；

```
p(f(a, g(h(b, X, Y), c), i(d, Z))):- ...
```

は；

```
p(f(a, T1, T2)):- T1 = g(T3, c), T3 = h(b, X, Y),
                  T2 = i(d, Z), ...
```

のようにコンパイルすれば，再帰的なユニフィケーションのためのスタックなどを使用する必要はない。

しかし，一般的なユニフィケーション，即ち二つのデータがどのようなものであるかコンパイル時には判定できない場合で，かつ双方が構造体である場合には，再帰的なユニフィケーションを実行時に行う必要がある。このために，何らかのスタックが必要となる他，一つのユニフィケーションに要する時間やメモリ量を予測できなくなることにも注意が必要である。

2.3 バックトラック

前述のように，バックトラックはAND-OR木の左優先／深さ優先の探索のための機構である。バックトラックはユニフィケーションの失敗によって引き起こされ，最後に通ったORノードへの後戻りを行う。その際に，ORの分枝に相当するクローズへの分岐とともに，実行環境をORノードを通った時の状態に復元する必要がある。

この実行環境の復元は，以下の二つの方法によって行われる。

(1) ORノードを通る際にその時点の状態を保存し，バックトラック時に保存した状態に戻す。

(2) ORノードを通った後で行った状態変更の履歴を記憶し，バックトラック時に履歴を逆にたどって元の状態に戻す。

これらの内，(2)は未定義変数の代入の無効化(Undo)に用いられ，それ以外の復元操作は(1)の方法を用いて行われる。以下，それぞれについて述べる。

2.3.1 状態の保存

状態の保存は，クローズのヘッド・ユニフィケーションの直前に行われる。この時点での実行環境は，基本的にはローカル・スタック及びグローバル・スタックが保持している。また，これ以後の状態の変化は；

- スタックの伸長
- スタック中の未定義変数への代入

である。未定義変数への代入については，後述するように履歴をとる方法で復元するため，状態の保存は二つのスタック・トップについて行えば良い。逆に言えば，バックトラックの際に二つのスタックを縮めることができる。この事実はグローバル・スタックにとって特に重要であり，意図的なバックトラックによってグローバル・スタック上に生成した構造体のための領域を解放することができる。この簡易的なガベージ・コレクションとも言える手法は，Prologのプログラムの中でしばしば用いられ，LISP等に比べて容易に記憶領域を節約／管理することを可能にしている*。

この他に保存しなければならない情報は，クローズが呼出された時点ではスタックに書かれていないものである。これらがどのような情報であるかは，実現方式に依存するが，2.2で述べた[Warren 83]場合；

*実行結果など忘れてはならない情報は，ファイル出力などの「副作用」によって記憶する。

(1) 引数レジスタ
(2) 呼出し側のEnvironmentへのポインタ
(3) 復帰アドレス

を保存しなければならない。また，バックトラック完了後に実行すべきORの分枝，即ち未実行のクローズの中の先頭のもの（Alternative Clauseまたは候補節）のアドレスも記憶しておかなければならない。

さて，一般には未実行の分枝を持つORノードは複数存在するため，バックトラックのために保存すべき情報も複数存在する。前述のようにOR木の探索方法は深さ優先であるので，複数の保存情報（Choice Pointという）の管理はスタックを用いて行うのが自然である。このスタックの実現方法は二種類あり，一つはChoice Point用のスタックを設ける方法であり[Carlsson 86]，もう一つはローカル・スタック上にChoice Pointを置く方法である。前者の方が簡単なように見えるが，スタックの数を増やすことによる管理オーバヘッドが発生する。また後者の方が，ローカル・スタックを縮める操作や，後述するローカル・スタック上の未定義変数のUndoのための操作などが，容易に行えると言う利点もある。この他，[Warren 77]やPSI-IのようにEnvironmentとChoice Pointを一つにまとめる方法などもあるが，ここでは[Warren 83]やPSI-II，PSI-IIIで採用しているローカル・スタック上にChoice Pointを独立に置く方法について議論を進める。

この方法では，Choice Pointの位置自体がORノード通過時のローカル・スタック・トップであるので，Choice Point中にはローカル・スタック・トップを保存する必要はなく，最新のChoice Pointのベースをレジスタ**B**に保持すれば良い*。なおレジスタ**B**は，スタック上のChoice Pointを結ぶために，Choice Point中に保存される†。

以上をまとめると，（Undoに関するものを除けば）Choice Pointに保存される情報は次のものとなる（図2-10）。

(1) 直前のChoice Pointのベース(**old-B**)
(2) Environmentへのポインタ(**E**)
(3) 復帰アドレス(**CP**)
(4) グローバル・スタック・トップ(**G**)
(5) 候補節のアドレス(**AP**)
(6) 引数レジスタ($\mathbf{A_n}$)

さて，バックトラックの際にはChoice Pointの情報が各レジスタにセットされるとともに，ローカル・スタック・トップはChoice Pointの直後まで縮められる。そして，Choice Pointに記憶した候補節(**AP**)を実行するが，その際さらに候補節があれば**AP**を次の候

*いずれの方式でもこの情報は必要である。
†いずれの方式でも同様。

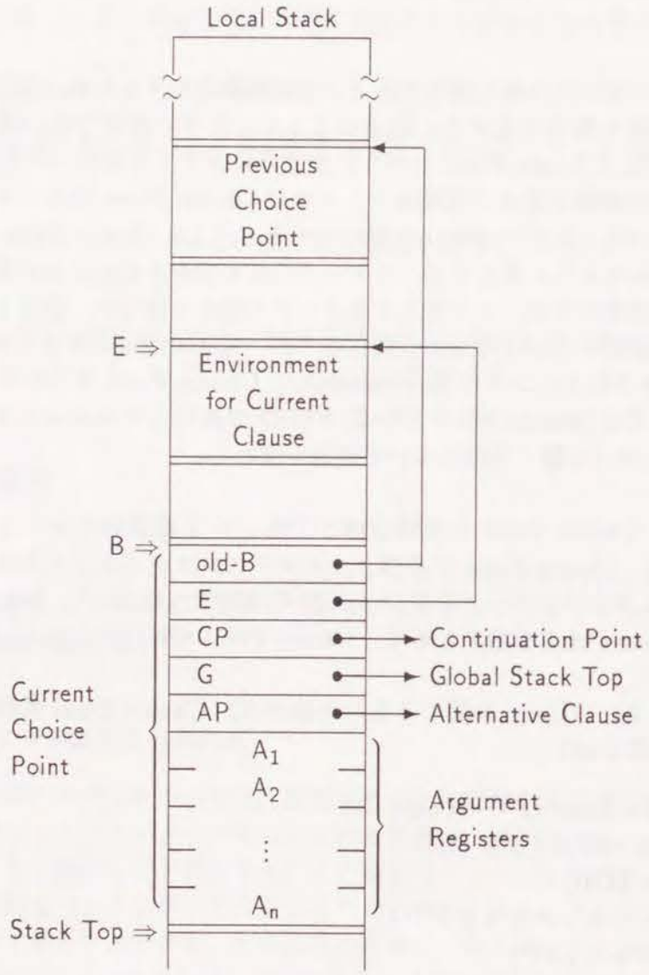


図 2-10: Choice Point

補節のアドレスに変更する。一方，**AP** が保持する候補節が最後のものであれば，Choice Point をローカル・スタックから取り除き，**B** を直前の Choice Point のベース **old-B** に設定する。

この Choice Point の除去が，最後の候補節の実行前に行われることに注意すべきである。即ち，クローズが一つしかない「決定的」な述語については Choice Point を生成する必要がなく，「非決定的」なものに比べて処理の手間が小さくて済む。このような述語は稀であるように見えるが，後述する最適化手法により，複数のクローズからなる述語を決定的であるかのように実行することができる。

なおここで，Environment の除去と Choice Point の関係について述べておく。2.2 において，クローズの実行完了時に Environment を除去すると述べたが，これはクローズの実行が決定的に完了した場合に限られる。即ち，クローズをルートとする AND-OR 木に未実行の分枝を持つ OR ノードがある時には，クローズの実行完了後にクローズの「途中」へバックトラックする可能性がある。この「途中」の時点におけるスタックの状態は，当然クローズの Environment を含むものであるので，クローズの実行が完了しても Environment を除去することはできない。具体的な Environment 除去の判定基準は，除去すべき Environment と最新の Choice Point の位置関係によって定まる。即ち，Environment が最新の Choice Point よりもスタックのボトム側にあれば $(E < B)$ 除去することはできず，そうでなければ $(E \geq B)$ 除去することができる。

2.3.2 Undo と Trail

未定義変数の代入の無効化は，OR ノードの通過後に行った代入操作の履歴を保存することによって実現される。この履歴保存は，LISP での Shallow Binding に似た方法で行われるが，Prolog の場合は変数への代入が一度しか行われないため，LISP に比べて容易である。即ち，LISP では変数のアドレスと旧値の双方を記憶しなければならないが，Prolog では変数のアドレスのみを記憶すれば良い。これをトレイル操作という。

無効化すべき変数のアドレスは，代入を行う際にトレイル・スタックというスタックにプッシュされる。バックトラック時にはトレイル・スタックを一つずつポップ・アップして，記憶したアドレスの変数を未定義状態に戻す Undo 操作を，バックトラック対象の OR ノードを通過した後に代入した変数に対して繰り返して行う。従って，OR ノード通過時のトレイル・スタック・トップ (**TR**) を記憶する必要があるが，これは Choice Point に保持される*。

*Choice Point に保持すべき情報は，これで全てである。

さて，変数への代入時に「常に」トレイルを行っても良いが，バックトラックによって「消滅する」変数に関するトレイルは無駄である。即ち，最新の Choice Point よりもトップ側にある局所変数や，Choice Point に保持されたグローバル・スタック・トップよりもトップ側にある大域変数は，バックトラック時のスタック縮退により消滅してしまうため，トレイル操作は不要である。実際，変数代入の半数以上がトレイル不要のものであるという統計があり [Touati 87]，また長い目で見ればほとんど全ての変数代入が決定的なものとなる [Nakashima 90a]。従って，トレイル・スタックの無駄な伸長を防止するためにも，選択的なトレイル操作を行うのが普通である。

更に，局所変数については最新の Choice Point よりもトップ側のものは，クローズの実行完了時の Environment 除去により消滅してしまうことに注意が必要である。即ち，このような変数のための領域は，メモリ管理の方法によっては，他の用途（例えば他のスタック）に用いられている可能性がある。そこで，バックトラック時にこのようなセルを不用意に Undo することができず，例えば変数アドレスとスタック・トップとの比較による選択的な Undo 操作が必要となる。従って，変数アドレスの比較がいずれにせよ必要であるため，トレイル・スタックを伸ばさない選択的トレイルが有利であることは明らかである。

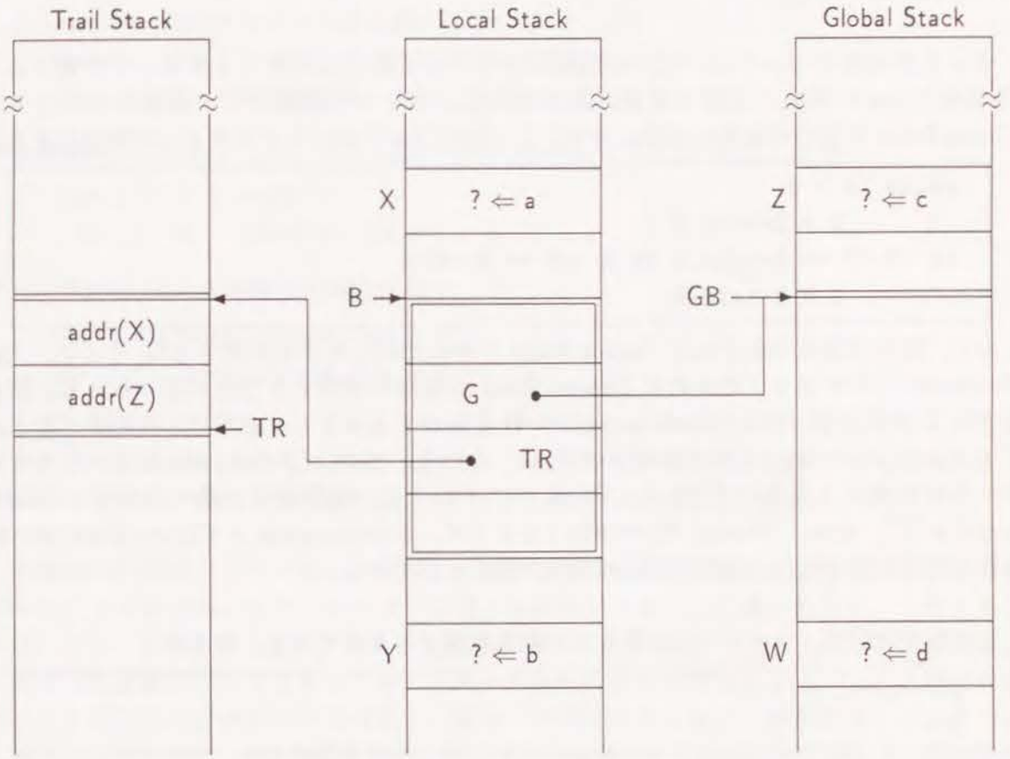
局所変数に関するトレイルの要否の判定は容易であり，変数が最新の Choice Point よりもトップ側にあれば，即ち変数のアドレスを A とした時に $A > B$ であればトレイルは不要である。大域変数に関しては，最新の Choice Point が保持するグローバル・スタック・トップとの比較が必要となるが，この値をレジスタ (**GB**) にキャッシュして高速化を図るのが普通である（図 2-11）。しかしこの場合，Choice Point を除去する際に **GB** を更新する必要があり，このためにかえってメモリ・アクセスの回数が増加することがあるという報告もある [Touati 87]。なお，トレイルの要否判定のためには，代入を行う変数がローカル／グローバルのどちらのスタック上にあるかを知る必要があることにも注意が必要である。

2.3.3 カット

バックトラックに関連する重要な操作の一つに，カットがある。カットは文法的にはゴールの一種であり，記号 '!' で表現されるが，明示的に OR 分枝を除去するという特殊な機能を持っている。即ちカット操作は；

(a) カットが含まれるクローズと「兄弟関係」にある OR 分枝
(b) カットよりも左側にあるゴールが生成した OR ノードが持つ分枝

を除去する。例えば；



$X < B \Rightarrow trail$
$Y > B \Rightarrow not\ trail$

$Z < GB \Rightarrow trail$
$W > GB \Rightarrow not\ trail$

図 2-11: トレイル

```
   p :- q, !, r.        q :- s.          s.
×  p :- ...          △  q :- ...      △  s :- ...
×  p :- ...          △  q :- ...      △  s :- ...
     ⋮                    ⋮                ⋮
×  p :- ...          △  q :- ...      △  s :- ...
```

では，'×'を付した分枝（クローズ）は(a)により，また'△'を付した分枝は(b)により，それぞれ除去される。

カットの実現については，三つの問題がある。まず第一は，カットによって不要となるChoice Pointの除去の方法である。具体的には，カットの結果として最新のものとなるChoice Pointを見つけなければならない。これは以下のアルゴリズムによって実現できる。

```
while (B > E)
        B = B->old_B ;
if (B->E == E->old_E && B->CP == E->CP)
        B = B->old_B ;
```

しかし，この方法は除去されるChoice Pointの数に比例した手間を要する*。そこで，Environmentの中にカットのためにChoice Pointの情報を保持する方法が用いられる。例えばPSI-IIやPSI-IIIでは，Environmentに**B**を保持するエントリを設け，そのタグによってカットによって除去されるか否かを判別している。また，[Debray 86]などではカットされた時に最新となるべきChoice Pointのアドレスを，局所変数の形で保持する方法をとっている†。なお，[Warren 77]やPSI-Iのように，EnvironmentとChoice Pointが一体になっている場合には，比較的容易に実現することができる。

二つ目の問題は，ローカル・スタックの縮退に関するものである。例えば；

```
p:- q, !, p.
p:- ...
```

の場合，Choice Point及びEnvironmentは以下のような順序で生成／除去される。

(1) pに関するChoice Point生成
(2) pの第一クローズに関するEnvironment生成
(3) カットによるChoice Point除去
(4) pの実行完了によるEnvironment除去

即ち，生成したChoice Point及びEnvironmentは全て除去されるため，ローカル・スタックのサイズがpの実行前と実行後で変化しないことが期待される。しかし，これを実現す

*全体の処理時間のオーダが変わるわけではない。
†この両者の優劣を判定するのは極めて困難である。

るのは必ずしも容易ではない。まず(3)では，除去されるChoice Pointよりも実行中のクローズ（pの第一クローズ）のEnvironmentがトップ側にあるため，ローカル・スタック・トップはEnvironmentの直後までしか縮めることができない。また(4)では，Environmentの除去，即ちEnvironmentの先頭までローカル・スタックを縮めることはできるが，除去されたChoice Pointの先頭まで縮めるのは困難である*。PSI-II, PSI-IIIでは前述のEnvironment中の**B**を保持するエントリのタグに，除去されてしまったChoice Pointが存在することを示す情報を持たせることによって，この問題を解決している。即ち，このエントリのタグは，指示しているChoice Pointが；

case-a： 初めからカットの対象ではない。
case-b： カットの対象であり，まだカットされていない。
case-c： カットの対象であったが，既にカットされた。

の3状態のいずれかとなる。カット操作ではこのエントリを`E->B`とした時；

case-a : `B = E->B`
case-b,c: `B = (E->B)->old_B`

とし，Environmentの除去の場合には；

case-a,b: `Local_Stack_Top = E`
case-c : `Local_Stack_Top = E->B`

として，除去されたChoice Pointがローカル・スタック中に残らないようにしている。なお，Environment用とChoice Point用にスタックを分離する方法や，EnironmentとChoice Pointを一体化する方法では，スタックの縮退は比較的容易である。

三番目の問題は，トレイル・スタックに関するものである。前述のように，選択的トレイルによりトレイル・スタックには「不用」な変数アドレスは記憶されない。しかしカットは，トレイル時には「有用」であった変数アドレスを「不用」にしてしまうことがある。このような変数アドレスをトレイル・スタックから除去する操作はTidy-Trailと呼ばれるが，これを行うか否かについては若干の議論の余地がある。即ち，選択的トレイルについてはトレイル・スタックを伸ばさないと言うだけではなく，Undoでのアドレス比較が不要となるため，無条件トレイルに対して有利であることが明らかであった。しかしTidy-Trailに関しては，トレイル・スタックから除去されない変数アドレスに対して，アドレスの比較とトレイル・スタック中での移動が必要であるため，スタックのアクセス回数や比較の回数については，選択的Undoの方が有利になる。従って，Tidy-Trailの効果はトレイル・スタックを伸ばさないことのみであることになる。

この効果の重要性は，キャッシュ・メモリのヒット率向上，物理記憶の割付回数の削減，更にはガベージ・コレクションの抑止など，アクセスの局所化に関係するものであり，定量

*そのようなChoice Pointの有無をコンパイル時に判断するのは，後述するClause Indexingとの関係で極めて難しい。

的に評価するのはかなり困難である。PSI系列の処理系ではアクセスの局所性を重視してTidy-Trailを行っているが，例えばトレイル・スタック・トップが一定の境界を超えるごとに，一括してTidy-Trailを行う方法なども有力であることを付記しておく。

なお，Tidy-Trailの対象となるのは，除去されるChoice Pointの中で最も古いものが保持するTR（これを TR_1 とする）からトップ側である。しかし，このようなChoice Pointを見つけるのは容易ではないため，カットによって最新となるChoice Pointが保持するTR（TR_0 とする）からトップ側を対象とすることが多い。この場合 TR_0 から TR_1 までについての操作は無駄であるが，操作を行うこと自体は問題とはならない。但し，PSI-II，PSI-IIIでは，カットが含まれるクローズが属する述語に関するChoice Pointがある場合には（前述の *case-b*），そのTRが TR_1 であることを利用して，無駄な操作を省いている。この省略が可能であるのも，EnvironmentにBを保持する利点の一つである。

2.4 基本的な最適化

2.4.1 Tail Recursion Optimization

Prologが持つ順序制御機能は，一種のサブルーチン・コールであるゴールの呼出しと，バックトラック（及びそれを制御するCut）のみである。従って，手続型言語における *for*, *while* などのループは，ゴールの再帰的呼出しにより実現される*。例えば；

```
while (condition)
        do_something ;
```

のような *while* ループは；

```
while_loop(...):- condition, !,
        do_someting, while_loop(...).
while_loop(...).
```

のように実現される。このように，最後に自分自身を呼出す形のループはTail Recursionと呼ばれ，*call* ではなく *goto* で実現できることが知られている。即ち，最後のゴールを呼出した後は，呼出し元のクローズに戻ってきても「何もすることがない」ので，最後のゴールは呼出し元のクローズ自身の戻り先へ，直接戻ることができる。具体的には，最後のゴールを呼出す前にクローズの実行継続情報であるEnvironmentを除去し，EnvironmentのベースEと復帰アドレスCPをクローズが呼出される前の状態に戻した上で，最後のゴールへの *goto* を行う。従って前述の `while_loop` では，自分自身を呼出す前にローカル・スタックが元の状態に戻るため，スタックを伸ばすことなくループを実行することができる†。これをTail Recursion Optimization (TRO) と言い，[Warren 80] により提案された後，ほとんどの処理系で採用されている。なお，この最適化は最後の呼出しが自分自身でない場合にも適用されるのが普通である‡。

TROはPrologに限った最適化手法ではなく，例えばLISPなどでも用いられている。しかし，Prologではユニフィケーションという強力な機構を用いて，LISPなどでは再帰呼出しの後に行われる処理を，呼出し前に実行することができることが多く，より効果的であると言える。例えば二つのリストを結合する `append` は，LISPでは；

*副作用を伴うバックトラックによりループを実現することもある。

†*condition* や *do_something* の実行途中では一時的にスタックが伸びることはある。また，*do_something* は決定的でなければならない。

‡このためLast Goal Optimization (LGO) と呼ぶこともある。

```
(defun append (l1 l2)
    (if (nil l1)
        l2
        (cons (car l1) (append (cdr l1) l2)) ))
```

のように，append を再帰的に呼出した後で cons が呼ばれるため，Tail Recursion とはならない。一方 Prolog では：

```
append([], L, L):-!.
append([X|L1], L2, [X|L3]):- append(L1, L2, L3).
```

のように，cons に当る操作をヘッド・ユニフィケーション [X|L3] で実行できるため，Tail Recursion とすることができる。

TRO に関しては，三つの注意すべき問題がある。第一はバックトラックとの関係であり，2.3 で述べたように最終ゴール以外のゴールが非決定的である場合，即ち最終ゴールを呼出す時点で $E < B$ である場合には Environment を除去することはできない。但し，**E** と **CP** の復元は無条件に行われる。また，Cut との関係も 2.3 で述べたものと同様である。

第二は，ゴール引数の受渡し方法である。2.2 で述べたように，ゴール引数を渡す方法の一つに，復帰アドレスの直後にゴール引数を評価するための情報を置いておく方法がある [Warren 77]。しかし，この方法は最終ゴールのヘッド・ユニフィケーションの際に呼出し元の Environment を必要とするため，TRO を適用することができない。従って TRO のためには，ゴール引数を呼出し前に評価し，スタックやレジスタに置く方法を用いなければならない。

第三は，最終ゴールに渡す局所変数の取扱いである。即ち，最終ゴールを呼出す前に Environment を除去するため，局所変数を最終ゴールが直接参照することはできない。そこで，局所変数をグローバル・スタックに割付ける，大域化 (Globalization) という操作が行われる。これには静的な方法と，動的な方法の二つがある。静的な方法では，最終ゴールに出現する変数をコンパイル時にグローバル・スタックに割付けてしまう。一方動的な方法では，最終ゴールを呼出す際に変数が未定義である場合にのみ*，変数をグローバル・スタックへ移動する [Warren 83]。筆者らの場合，PSI-I では静的な方法を用いたが，PSI-II，PSI-III ではグローバル・スタックの伸びを最小化するために動的な方法を用いている。なお，いずれの方法においても変数がヘッドまたは構造体中に出現している場合には，変数セルそのものが Environment 中には存在しないため大域化の必要はない。

*正確には未定義変数セルそのものが除去しようとしている Environment 中に存在する場合であり，極めてまれである。

さて，クローズが一つしかゴールを持たない場合 (Transitive Clause)，TRO により Environment の生成が不要となるという，極めて効果的な最適化を行うことができる。例えば，復帰アドレス **CP** に関しては，クローズが呼出されてから実行完了（ゴールへの *goto*）までの間その値は変化しないため，Environment への退避と復元は不要である。問題は局所変数であるが，これらは全てレジスタ上に置くことができる。例えば append の第二クローズ；

```
append([X|L1], L2, [X|L3]):- append(L1, L2, L3).
```

の第一，第二引数がリスト・セルで，第三引数が未定義変数の場合*，変数 X, L1, L2, L3 をレジスタ $R_1 \sim R_4$ にそれぞれ割当てて，以下のように処理することができる。

(1) 第一引数がリスト・セルであることを確認し，その car を R_1 (X) へ，cdr を R_2 (L1) へ代入する。

(2) 第二引数を R_3 (L2) へ代入する。

(3) 第三引数が未定義変数であることを確認し，グローバル・スタック上にリスト・セルを生成する。その car には R_1 (X) を代入し，cdr は未定義変数として R_4 (L3) にそれへの Reference Pointer をセットする。

(4) R_2 (L1), R_3 (L2), R_4 (L3) を，ゴールの第一，第二，第三引数として引渡す。

ゴール引数を引数レジスタ $\mathbf{A_n}$ にセットする方式では，更に大きな効果を得ることができる。即ち，L1, L2, L3 を $\mathbf{A_1}$, $\mathbf{A_2}$, $\mathbf{A_3}$ に割当てることにより，(4) の操作が全く不要となる。特に L2 については (2) の処理も不要となり，第二引数に関する操作は完全に除去される。

以上のように Transitive Clause は最適化に関して大きな意味を持っているが，その出現頻度も見かけよりはかなり大きい。例えば前述の；

```
while (condition)
        do_something ;
```

において，*condition* や *do_something* を，append のようにヘッド・ユニフィケーションのみで実現できる場合が少なくない。更に PSI 系列の処理系では，後述するように様々な機能を持った組込述語の実行を，ヘッド・ユニフィケーションの一部とみなすことにより，最適化の適用範囲を大きく広げている。

なお，ゴールを持たないユニット・クローズについても，当然同様の最適化が可能である。また，局所変数のレジスタへの割付や，引数レジスタの効率的な利用については，一般のクローズでの第一ゴールの呼出しにも適用して，高い効果を得ることができる。このため，この最適化は First Goal Optimization (FGO) と呼ばれる。

*これが普通である。

この他 TRO に類似した最適化として，[Warren 83] ではクローズの実行途中で，不要になった局所変数を Environment から除去する方式が提案されている。例えば；

```
p:- q(X), r(X, Y), s(Y, Z), t(Z).
```

では，局所変数を Environment 中で Z, Y, X の順に配置し，r の呼出し前に X を，また s の呼出し前に Y を，それぞれ Environment から除去する（ローカル・スタックを1ワードずつ縮める）。但しこの方法では，除去対象の変数の大域化の要否の判定や*，Choice Point との関係によるスタック縮退の可否の判定が増加するため，その効果はかなり疑問である。

2.4.2 Clause Indexing

Prolog では手続型言語における *if-then-else* や *switch-case* のような条件判定，また *for, while* などのループの終了判定は，全てユニフィケーションとバックトラックによって行われる。しかしバックトラックの処理は状態の復元を伴うため，手続型言語での判定処理に比べてかなり「重い」処理である。そこで，引数（通常は第一引数）のデータ型や値による「軽い」条件分岐を用いて，バックトラックをできるだけ行わずにクローズを選択する Clause Indexing という最適化手法が用いられる。

例えば；

```
capital(北海道, 札幌).
capital(青森, 青森).
capital(岩手, 盛岡).
        ⋮
capital(沖縄, 那覇).
```

という述語を，ゴール capital(沖縄, X) によって呼出すと，単純に処理した場合 46 回のバックトラックの後，クローズ capital(沖縄, 那覇) が実行される。この単純な実現手法には；

- 一回のバックトラックが「重い」
- バックトラックの回数が多い

という問題点がある。

まず前者を解決するためには，第一引数と特定のアトムの比較による条件分岐命令の導入が考えられる。例えば [Warren 77] の命令；

*変数が「完全に」不要になってから除去すれば（例えば X を s の呼出し前に除去する）この判定は不要である。


```
ifatom  A, Label
```

は，第一引数がアトム A に等しければ Label に分岐し，そうでなければ次の命令を実行する。これを用いて；

```
ifatom  北海道, capital_hokkaidou
ifatom  青森, capital_aomori
ifatom  岩手, capital_iwate
        ⋮
ifatom  沖縄, capital_okinawa
```

という命令列を設ければ（capital_x は x県のクローズのためのコードとする），ゴール capital(沖縄, X) の場合，47 回の ifatom という「軽い」命令の実行によりクローズ capital(沖縄, 那覇) を選択できる。

バックトラックの回数を減らすためには，ハッシングの手法が用いられる。例えば命令；

```
hash_by_jatom  Label_a, Label_i, ..., label_n
```

が第一引数が漢字アトムであるとして，その日本語読みが「あ」「い」...「ん」のいずれで始まるかによって Label_a, Label_i, ..., Label_n に分岐するものであるとする*。これを用いて；

```
hash_by_jatom   captial_a,          ; あ：青森，秋田，愛知
                captial_i,          ; い：岩手，茨城，石川
                fail,               ; う
                capital_ehime,      ; え：愛媛
                capital_o,          ; お：大阪，岡山，大分，沖縄
                    ⋮
                capital_wakayama,   ; わ：和歌山
                fail                ; ん
```

とし，ラベル capital_o に命令列；

```
ifatom  大阪, capital_oosaka
ifatom  岡山, capital_okayama
ifatom  大分, capital_ooita
ifatom  沖縄, capital_okinawa
```

を置けば，ゴール capital(沖縄, X) の実行は極めて高速化される。

また，[Warren 83] では第一引数が未定義変数，アトミック，リスト，複合項のいずれかによって四方向に分岐する命令が提案されている。これは append のようなリストか nil による分岐や，数式処理や構文解析を行う述語に対して，極めて有効である。

*実際はアトム番号の下位ビットなどが用いられる。

さて，Clause Indexingが最も効果を発揮するのは，第一引数の判定により選択可能なクローズが一つに絞られる場合である。`append`や`capital(沖縄, X)`はこの範疇であり，バックトラックを全く起こさずにクローズを選択できる。のみならず，選択肢が一つしかないため，2.3で述べたようにChoice Point生成も不要であり，その実行は極めて高速に行うことができる。

一方，Clause Indexingによっても，クローズを一つに絞れない場合も少なくない。例えば第一引数が未定義変数の場合，全てのクローズが選択対象となる。この可能性を排除するのは極めて困難であり，`append`のような述語に対しても，第一引数が未定義変数である場合に対処するためのコード，即ちChoice Pointの生成/除去のためのコードが必要となる。また，ヘッドの第一引数が同じであるようなクローズが複数ある場合には，それらの中からの選択はバックトラックにより行わなければならない。更に，ヘッドの第一引数が変数であるクローズが存在すると，そのクローズはいかなるゴール引数に対しても選択可能であるため，Choice Pointの生成は必須となる。

なお，Clause Indexingを第一引数以外の引数や[Kurosawa 88]，リスト／複合項の要素を対象として行うことも考えられる。また，データ型を判定する組込述語がクローズ中にある場合，それをClause Indexingの中に取り込む手法も提案されている[Carlsson 87]。

2.5 ESP/KL0

ESP [Chikayama 83a, 84] はPSI系列の逐次型推論マシンの基本言語であり，そのオペレーティング・システムであるSIMPOS [Yokoi 84] を含めた全てのプログラムはESPで記述される。これらの大規模かつ実用的なプログラムの開発を容易にするために，ESPではPrologに対する様々な拡張がなされている。その中でも最も重要なものは，オブジェクト指向の概念の導入と，実行順序制御機能の強化である。

オブジェクト指向の概念は，Prologの欠点の一つである「平板さ」を解決するために導入されたものである。Prologの述語は，構文及び意味の両面において互いに対等な関係にあり，関連を持った述語の集合としての「モジュール」や，述語間の「上下関係」といったものは存在しない。従って，個々の述語にユニークな「名前」を与えるのが難しいばかりではなく，全ての述語が「公開」されてしまうという欠点がある。

また，Prologが持つ単一代入と全ての変数がローカルであるという性質は，「状態」の実現を困難にしている。即ち，「状態」を保持する変数を引数（またはその一部）として，直接関与する述語ばかりではなく，それらを結び付ける全ての述語に与えなければならない。

これらの性質は，大規模なプログラムの開発には極めて不便である。例えば，ファイル・システムをPrologで実現する場合，open, close, read, writeといったインタフェースとなる述語だけでなく，ディスクの起動を行う述語のような秘匿すべきものも公開されてしまう。また，ディスクの物理的なブロック位置のような状態変数を，readやwriteの引数として公開しなければならない。このように極端に「開放的な」システムは危険であると同時に，細部の些細な変更が外部に伝播するのを抑止することが極めて困難であるという重大な欠点を持っている。

これらの問題点は，例えばCommon-LISPのPackage，グローバル変数，破壊的代入などの「汚い」手段によっても解決することはできる。しかし，ESPではオブジェクト指向という枠組みを用いることにより，「美しい」解決が可能となったばかりではなく，インスタンスや継承といった極めて有用な機能をも提供することができたのである。

ESPのプログラムは『クラス』と言うモジュールを単位として記述され，その中で外部に「公開」される述語である『メソッド』と，外部から「秘匿」された『ローカル述語』の二種類が定義される。また，クラスを「鋳型」としてその「インスタンス」である『オブジェクト』を生成することができる。オブジェクトは固有の状態変数である『スロット』を持つ一方で，メソッドやローカル述語は同じクラスに属する他のオブジェクトと共有している。このスロットへのアクセスは「名前」を用いて行うことができ，また公開／秘匿の選択も可能である。この他，類似した機能を持つクラスを作成する際に，その共通部分を「親ク

ラス」として定義し，「子クラス」では固有の追加機能だけを定義する「継承機能」や，継承した機能の一部分を変更する「デーモン」など，ソフトウェアのモジュール性を高めるための機能が用意されている。更にESPには，柔軟でかつ高い機能をもったマクロの機構が備えられており，Prologに比べて一層書きやすい言語となっている[Kondoh 88]。

一方，実行順序制御機能はエラー処理などの，実用的なプログラムには欠かせない機能を容易に実現するために導入された。例えば，ユーザが作成したESPプログラムのバグにより，数値ではない値の加算が実行されようとした時，プログラムの実行を管理するESP-Listener（これもESPプログラムである）は以下のような処置をとらなければならない。

(1) エラーの検出
(2) ユーザへの通知
(3) openされたファイルのcloseなどの「後始末」
(4) トップ・レベルへの大域脱出

これらの処理を，Prologがもつ実行順序制御機能，即ち述語の呼出し，バックトラック，及びカットのみで実現するのは，ほとんど不可能である。そこで，以下に示すような強力な順序制御機能が，ESPの土台となっている言語であるKL0 [Chikayama 83b]に導入された。

- 組込述語によるエラー検出と「例外ハンドラ」の呼び出し。
- バックトラックを用いた大域脱出を可能とする「遠隔カット」。
- バックトラック時に実行される述語を定義する *On-Backtrack*。
- 未定義変数への代入時に実行される述語を定義する *Bind Hook*。

この他，順序制御機能の以外の組込述語として，データ・タイプ判定，構造データ操作，算術／論理演算及び比較，更にはオペレーティング・システムのサポートなど，150種以上が用意されており，全てがマイクロプログラムによって実現されている。

以上述べたESPのオブジェクト指向機能やKL0の実行順序制御機能の実現方式は，処理系全体の性能を大きく左右するものである。具体的な方式は各々の機能の仕様とともに付録Aに示すが，最適化との関係などについては4.3で詳しく論ずる。

2.6 KL1

KL1 [Miyazaki 88, Chikayama 88]は，並列推論マシンのために開発された並列論理型言語であり，その言語仕様はGHC (Guarded Horn Clauses) [Ueda 85, 86a]という論理型言語に基いている。GHCはCommitted Choice Languageと呼ばれる言語の一種であり，このグループに属するPARLOG [Clark 86]やConcurrent Prolog [Shapiro 86]などと同じく，以下の性質をもっている。

(1) AND関係にあるゴールは並列に実行される。

(2) ゴール間の同期は共有変数のユニフィケーション，具体的には未定義変数の代入により行われる。

(3) 全てのクローズは「コミット」と呼ばれるカットに類似したオペレータを持ち，一つのクローズだけが決定的に選択される。

これらの性質はPrologのような「普通の」論理型言語とはかなり異なっており，並行動作するプロセスの記述や，並列マシンのためのプログラミングを意識したものとなっている。また，プロセスをオブジェクト，プロセス間通信をメソッドの適用とみなすことにより，オブジェクト指向の機能を論理の枠組みの中で実現できることも特質の一つである[Yoshida-K 88]。実際，これらの特質を生かして，並列推論マシンのためのオペレーティング・システムPIMOS [Chikayama 88]や，様々な並列応用プログラムがKL1を用いて記述されている。

一方，言語処理の観点からは，並行動作するゴールのスケジューリング，同期のメカニズムなど，通常の言語にはない機能の実現が要求される。また，ガベージ・コレクションや，プロセッサ間でのユニフィケーションなどの実現手法も，処理系を構築する上で解決しなければならない重要な課題である[Nakajima 89]。特に，MRB (Multiple Reference Bit)という1ビットのリファレンス・カウンタを用いた実時間ガベージ・コレクションの方式[Chikayama 88, Kimura 90]は，推論マシンのハードウェア・アーキテクチャに大きな影響を及ぼしており，3.3.4で述べるように特別なサポート機構がPSI-IIIにおいて導入されている。

なお，KL1の言語仕様と実現手法に関しては，付録Bにおいて概説しているので参照されたい。

第3章
推論マシンのアーキテクチャ

前章において，論理型言語の処理の様々な局面で，タグに関連する操作が行われることを述べた。また，構造データの動的な生成，複数のスタック操作，ガベージ・コレクションなど，記憶領域管理が他の言語とはかなり異なったものであることを明らかにした。更に，付録A及びBに示すように，ESP/KL0やKL1においてはPrologに比べてより動的な処理が必要であり，処理系の柔軟さが求められる。

これらの特徴，即ちタグ操作，記憶管理，処理の柔軟性は，筆者らが開発した推論マシンであるPSI-I, PSI-II, PSI-IIIの全てに共通した，基本的な設計方針である。

まずタグに関しては，図3-1に示すように，8ビットのタグを含む40ビット・ワードの構成とし，多様なデータの表現と充分なアドレス空間の提供の双方を満足できるようにしている。また，タグと値をハードウェア構成の上で分離してそれぞれの抽出を容易にするとともに，タグの付け替えや判定のための様々なハードウェアを用意している。更に，タグの中にガベージ・コレクションのためのビットを用意することによって，その効率的な実現に寄与している。

記憶管理については図3-2に示すように，個々のスタックに対応する「エリア」という概念を導入し，独立に伸縮するスタックの個別管理を実現した。また，ページ／オフセットによるアドレス変換や，動的な記憶領域割付のためのハードウェア・サポート，大容量のキャッシュ・メモリの実装など，高速なメモリ・アクセスへの手厚い配慮が成されている。記憶容量の面でも，16 ～ 64 Mw (80 ～ 320 MB) という極めて大容量の物理記憶の実装を可能とし，大規模なAI応用プログラムの実行に充分に対処できるようにしている。

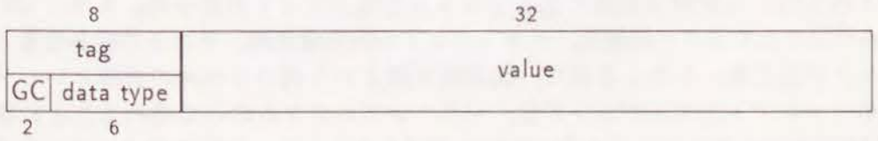


図 3-1: データの形式

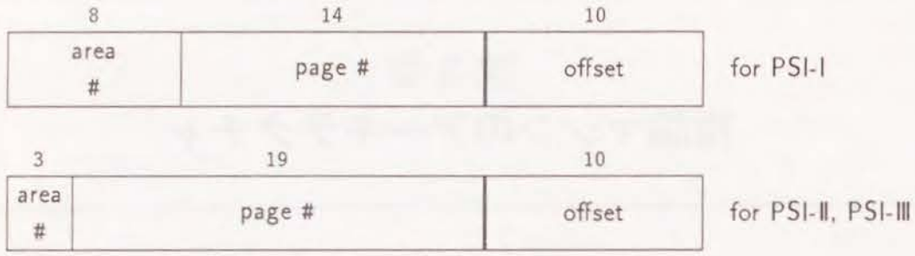


図 3-2: アドレスの形式

柔軟な処理系の構築という点では，マイクロプログラミング [Hagiwara 77] の手法を最大限に活用した。特に，大容量の WCS (Writable Control Store)，水平型のマイクロ命令，豊富なレジスタ／テーブル類など，高速性と柔軟性というマイクロプログラミングの特質に適合したハードウェア設計を行った。また，マイクロプログラムの生産性を考慮した，デバッグや評価のためのサポート機能も，特徴の一つとなっている。

これら，論理型言語の高速処理のためのハードウェアとともに，商用の AI ワークステーションとしての充分な実用性と信頼性も，ハードウェア・アーキテクチャの設計における極めて重要な要素である。図 3-3に示すようにいずれのマシンにおいても，固定ディスクなどの外部記憶装置，様々な標準インタフェース（TCP/IP, GPIB, SCSI, RS232C など），高機能コントローラを備えたビットマップ・ディスプレイなど，豊富な入出力装置が備えられている。また，低速入出力装置のコントローラを兼ねる Console System Processor には，マイクロプログラムやソフトウェアのデバッグ機能が備えられており，システム開発の効率化に大きく寄与している [Nakashima 84]。信頼性の面では，主記憶（物理記憶）の各ワードに対する ECC ビットの付加，各所に設けられたパリティ・チェック機構など，商用機として必要かつ充分な配慮を行っている。

一方，個々のマシンに関しては以下のような特徴がある。まず PSI-I では，KL0 の「内部表現」をマイクロプログラムが解釈／実行するという，「マイクロ・インタプリタ」方式を採用した。このマイクロプログラムに重点をおいた設計を行った結果，最適化手法の試行，処理系の挙動に関する評価など，言語処理の研究のための様々な実験を，ファームウェアという閉じた世界で実施できるという大きなメリットがあった。また，OS 機能のファームウェア化により，高速化，ソフトウェアの開発量削減，テストの容易化など，様々なメリットが生じた。しかしながら，論理型言語という高いレベルの言語とハードウェアとの間のセマンティクス・ギャップを，マイクロプログラムのみで埋めることは難しく，37.5 KLIPS* という当時の最高水準の性能を達成したものの，性能改善の余地がかなり大き

*Kilo Logical Inference per Second の略であり，一秒間に行われるゴール呼びだし回数を意味する。

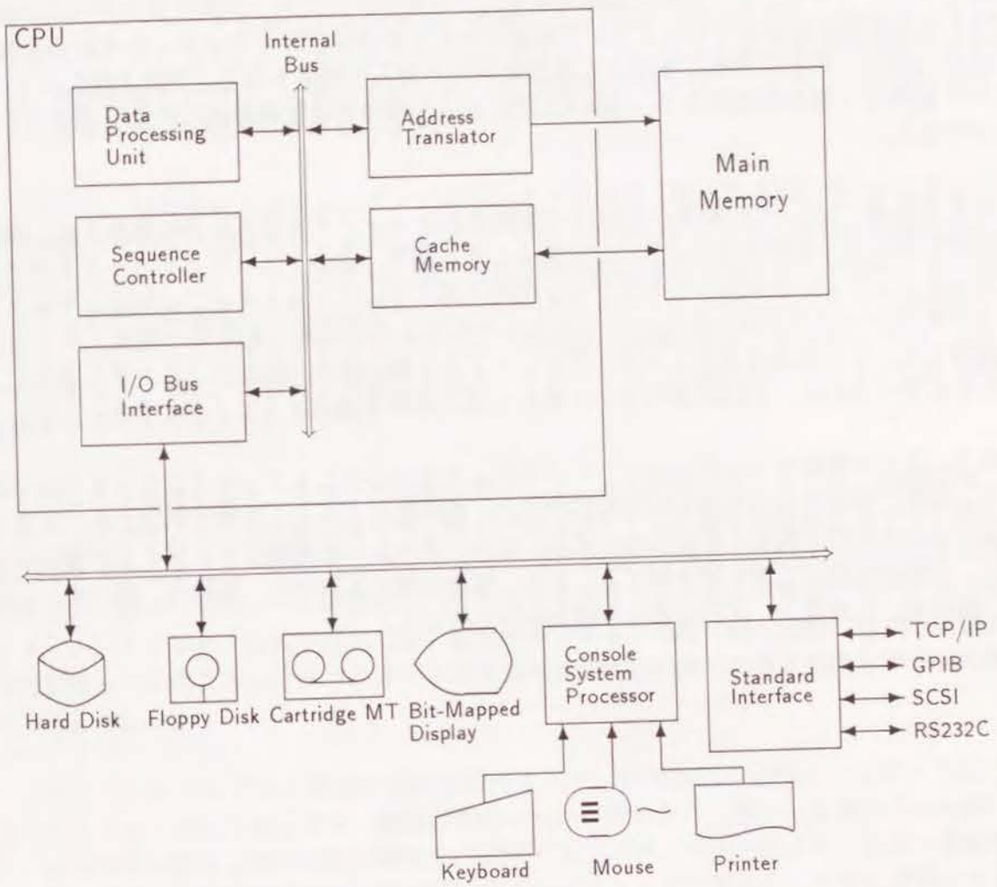


図 3-3: システム構成

いことが明かになった。

そこでPSI-IIでは，プログラムをWAM (Warren Abstract Machine) [Warren 83] と呼ばれる抽象マシンの命令へコンパイルし，この機械命令をマイクロプログラムでエミュレートする方式を採用した。この方式が性能面でマイクロ・インタプリタ方式に優ることは，PSI-I上にWAMの実験的な処理系を構築して，その評価を詳細に行うことによって確認した。また，ハードウェアの面では機械命令のフェッチ／デコードに関するサポートの他，PSI-Iでの評価に基づいてハードウェア機構やマイクロ命令アーキテクチャの大幅な改良を行った。更に，新たに考案した様々な最適化手法の導入の効果もあり，PSI-Iの3～11倍という極めて大幅な性能向上と，430 KLIPSという当時では世界最高の性能を達成することができた。

次にPSI-IIIでは，その効果がPSI-IIで実証されたコンパイル方式を更に発展させ，論理型言語処理に向いた命令パイプラインを導入した。即ち，命令フェッチ／デコードやオペランド・フェッチという従来のパイプライン機能に加えて，データ・タイプの判定とデレファレンス処理をパイプライン化し，ハードウェアによる並行処理の範囲を一層拡大する方式を採用した。この改良はマシン・サイクルとサイクル数の両面で性能向上に大きく寄与し，PSI-IIの2～3.5倍，1.5 MLIPSという極めて高い性能を達成することができた。

以下，各々の推論マシンについて，ハードウェアを中心としたアーキテクチャについて論ずる。特に，冒頭で述べた基本的な設計方針が，具体的にどのような形で実現され，またそれらがいかに利用されるかを明らかにする。なお，各マシンの性能などに関する評価については，一部分を除いて第5章で論ずる。また，本章に関する研究は，瀧和男，横田実，山本明，西川宏，中島克人，武田保孝らの協力を得て行った。

3.1 インタプリタ方式による逐次型推論マシン (PSI-I)

3.1.1 設計方針

PSI-I [Taki 84] は世界初の論理型言語専用プロセッサであったため，その設計段階においてはアーキテクチャや処理方式に関する，経験に基く指針が得られていたわけではなかった。従って，どのようなハードウェアが有効であるか，いかなる処理方式が最適であるかなどは，設計段階では明らかになっておらず，逆に設計や評価を通じてこれらの問題を研究することがPSI-Iの目的の一つでもあった。

一方では，第五世代コンピュータ・プロジェクトにおける様々なAI応用ソフトウェアの研究推進のために，開発用ツールとしてのAIワークステーションを，早急に提供すことが強く要望されていた。このためには，単に高速な推論機能を持ったCPUを供給するだけではなく，入出力機器などの周辺装置や，オペレーティング・システム，プログラム開発ツールなど，実用的なシステムに不可欠である様々な機能の提供が必要であった。また，OSなどの基本ソフトウェアの開発の効率化を図り，ソフトウェアを含めたシステム全体を早期に完成させることも求められていた。

このような背景に基き，PSI-Iの設計に当っては，本章冒頭で示したタグ，記憶管理，マイクロプログラムの三大方針の中でも，マイクロプログラムを最も重視することとした。即ち，ソフトウェアとファームウェア／ハードウェアの接点である機械命令を，KL0の内部表現という極めて高いレベルに設定し，それをマイクロプログラムが解釈／実行する「マイクロ・インタプリタ」方式を採用した。またハードウェア構成に関しては，特定の処理の高速化を狙った機構よりも，マイクロプログラムから汎用的に使用できる機構を重点的に装備することとした。

このマイクロプログラム重視の設計方針によって，言語処理の研究という面では以下のようなメリットが生じた。まず，機械命令のレベルが高いため，種々の最適化手法の試行など，処理方式に関する様々な試行錯誤を，ファームウェアという「閉じた」世界で実施することができた。即ち実現手法の変更を，ソフトウェアという「開いた」世界に波及させることなく実施できたため，このような作業にありがちな混乱を最小限のものとすることができた。また，評価のための計測などをマイクロプログラムの僅かな修正により，容易にかつ高速に行うことができたことも，処理方式の研究に大きく寄与した。

また，高レベルの機械命令，特にOSサポート用組込述語の導入により，従来ソフトウェアで行われていた処理がファームウェアに大きく委譲された。この結果，OSの開発量が削減されたばかりではなく，テストの容易化やバグの局所化などにより，高い信頼度を持ったシステムを迅速に開発することができた。また，実現方式の詳細やハードウェアとの具体的

なインタフェースを，プログラマに対して隠蔽したことにより，記憶管理などのOS機能を含めた最適化や，後継機への容易な移行が可能になったのも，見逃すことができないメリットである。

一方，ソフトウェアとハードウェアの間の極めて大きなセマンティクス・ギャップは，処理の効率の面では様々な不利益をもたらした。その結果，ソフトウェア・インタプリタを性能面で遥かに上回るマイクロプログラムの利用や，様々なハードウェア・サポートをもってしても，PSI-Iの性能は必ずしも満足できるレベルには到らなかった。しかしながら，設計当時の世界最高速の処理系であったDEC-2060上のPrologと同等の性能を達成することができ，上述のような多くのメリットがあったことを勘案すると，最初の論理型言語専用プロセッサとしての役割を充分に果たすことができたと考える。

3.1.2 機械命令アーキテクチャ

前述のように，PSI-Iの機械命令はKL0のプログラムを「内部表現」に変換したものであり，通常のものとはかなり異なった高レベルの形式となっている[Yokota 84]。例えば，*Quick Sort*プログラムの一部分である述語partition；

```
partition([],_,L,L):-!.
partition([X|L1],Y,[X|L2],L3):- X < Y, !, partition(L1,Y,L2,L3).
partition([X|L1],Y,L2,[X|L3]):- partition(L1,Y,L2,L3).
```

は，図3-4に示すように変換される。

図に示したように，内部表現はソース・プログラムのイメージに近く，変数の分類（局所か大域か）と番号付け，述語名のアドレスまたは組込述語コードへの変換などがなされている程度である。また，ヘッドやゴールの引数が何であるかや，どのようなゴールが存在するかはタグによって判別されるようになっている。例えば`lvar`, `gvar`はそれぞれ初出の局所／大域変数であり，`lref`, `gref`は局所／大域変数の二回目以降の出現に対応する。また，nil（'[]'）やリスト・セルのように，ソース・プログラムに出現したデータは，それらの内部表現自身がコード中に出現する*。更に，タグが`code`, `blt`であるデータは，通常のゴール及び組込述語ゴールの呼出をそれぞれ意味している†。

最も重要な処理であるヘッド・ユニフィケーションは，概念的には以下のように行われる。まず，タグ`code`を持つデータを「発見」すると，それが指示する述語の先頭に実行中クローズを指すポインタ**cLAR**をセットする。同時に，ゴール引数を指示するポインタ**pLAR**を`code`データの直後に設定する。述語の先頭部分には述語全体の情報（述語ヘッダ），及び第一クローズに関する情報（クローズ・ヘッダ）があるので，それらを適宜保存した後，**cLAR**は第一クローズの第一引数の位置に設定される。なお，図3-4には，`partition`の第三クローズから自分自身を再帰的に呼出した際の，**pLAR**と**cLAR**のこの時点での位置を示している。

次に，**pLAR**と**cLAR**が指示するエントリを読み，それぞれの内容に従ってユニフィケーションを行う。従って，この操作は2.2で述べた「一般的な」ユニフィケーションに近い処理となる。一つのユニフィケーションが完了すると，**pLAR**と**cLAR**をインクリメントし，次のユニフィケーションを行う。これを引数の数（述語のヘッダに記されている）だけ繰り返すとヘッド・ユニフィケーションが完了し，最初のゴールの呼出し処理を開始する。一方，ユニフィケーションが失敗すると，クローズ・ヘッダに従って次のクローズへ

*PSI-Iでは，リスト・セルのコード中の正しい表現はcdrを第一要素，carを第二要素とするヒープ・ベクタであるが，説明を簡単化するために`list`タグを持つデータとしている。

†実際にはクローズの最終ゴール呼び出しは，`lcode`, `lblt`をタグとするデータが行なう。

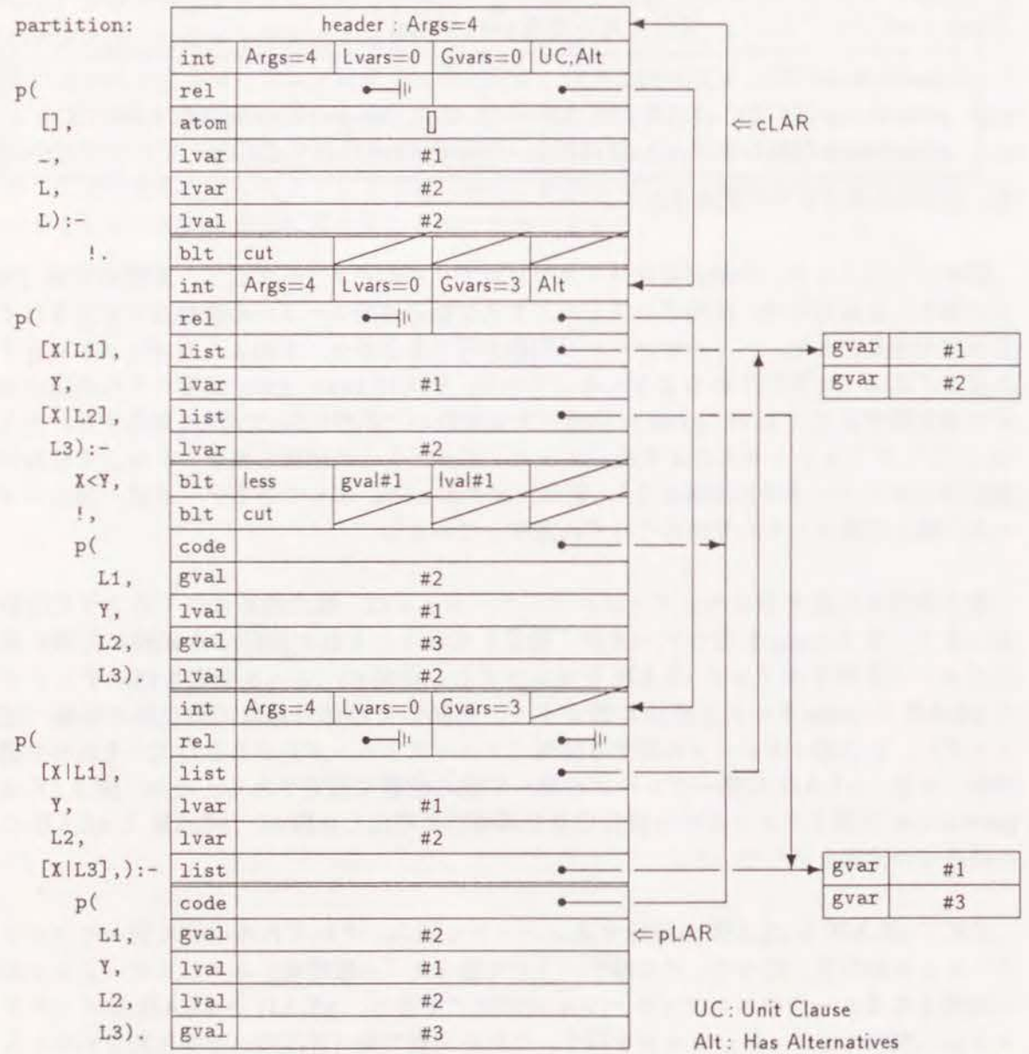


図 3-4: KL0 の内部表現


```
nth(N,[_|L],E):- N1 is N-1, nth(N1,L,E).
```

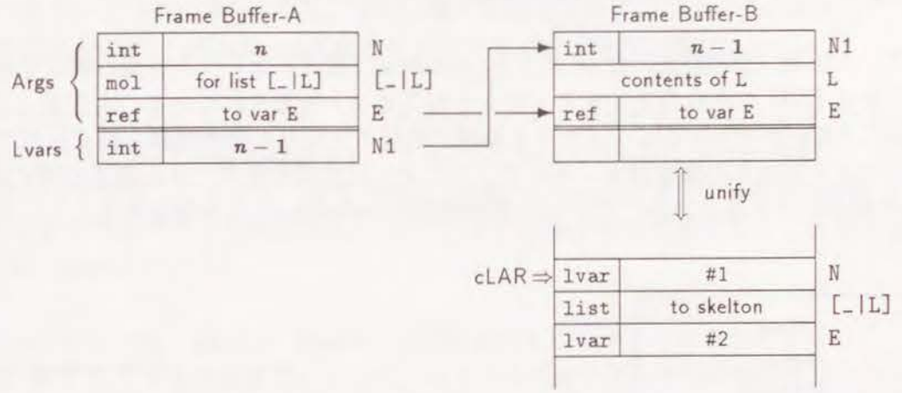


図 3-5: フレーム・バッファ

cLAR を移動し，また **pLAR** をヘッド・ユニフィケーション開始時の値に戻して，再度ユニフィケーションを行う。

以上のような「概念的」手法は [Warren 77] に類似したものであり，2.4 で述べたように Tail Recursion Optimization を適用することができない。また，ヘッド・ユニフィケーションの失敗によって次のクローズへ移行する *Shallow Backtrack* のたびに，**pLAR** が指示する引数を繰返し評価する必要もある。そこで，ゴールの呼出に先立って，「フレーム・バッファ」という高速バッファに引数の評価値を設定する，「引数コピー方式」が採用された [Yokota 84]。

フレーム・バッファは図 3-5に示すように，引数の評価値と局所変数を格納するバッファであり，後述するように高速 RAM によって実現されている。従って，前述のヘッド・ユニフィケーション処理は，**cLAR** が指示するヘッド引数と，フレーム・バッファに格納された評価値とのユニフィケーションとなり，かなりの高速化を期待することができる。

また First Goal Optimization のために，二つのフレーム・バッファが設けられている。即ち，第一ゴールの呼出の際には，バッファ間の移動によって引数を準備することができ，これについても高速化が期待できる。

なお，フレーム・バッファを真に活用するためには，順次アクセスとランダム・アクセスの双方のサポート，二つのバッファの高速な切換などが必要である。これらに対応するためのハードウェア機構については，3.1.4 において述べる。

3.1.3 ハードウェア構成

PSI-Iのハードウェアは，図3-6に示す構成となっている。

まず，主記憶（物理記憶）は40ビット×16 Mw（最大）という大容量の構成であり，各ワードにはECCのためのチェック・ビットが7ビット付加されている。主記憶とCPUの間には，8 Kwのキャッシュ・メモリと論理／物理アドレスの変換機構が備えられている。キャッシュ・メモリの構成はセット・アソシアティブ方式であり，また書込の方式はStore Backを採用している。なお，アドレス変換機構については3.1.6で詳述する。

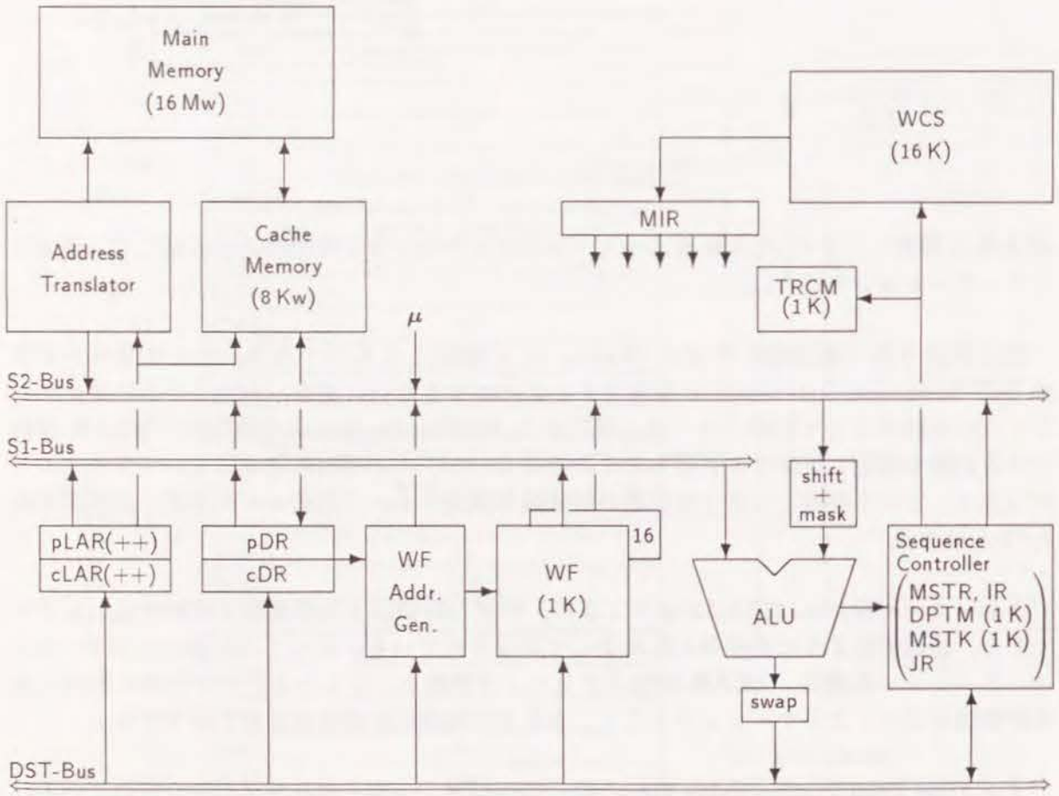


図 3-6: PSI-Iのハードウェア構成

メモリ系とCPUとのインタフェースは，二つのアドレス・レジスタ **pLAR** と **cLAR**，及び二つのデータ・レジスタ **pDR** と **cDR** を用いて行われる。このように，アドレス／データ・レジスタを二組設けたのは，論理型言語の処理においてはメモリ・アクセスが頻繁になされるという予測に基いている。また，様々な局面で成される主記憶の順次アクセスの高速化を図るために，**pLAR/cLAR** にはインクリメント機能が備えられている。

CPUの中核部であるALUには，二つの入力データ・バスであるS1-BusとS2-Bus，及び出力バスであるDST-Busが接続されている。S1-Busには **pDR**, **cDR** の他に，1 Kwの容量を持つレジスタ・ファイルである **WF** が接続されている。また，**pDR/cDR** と **WF** の先頭16 wについては，これらのレジスタに関する操作が高頻度であることを予想し，S2-Busにも接続されている。

S2-Busにはこの他，**pLAR/cLAR**，定数生成のためのマイクロ命令フィールド，3.1.4で述べる **WF** のアドレス生成用レジスタ，マイクロプログラムの順序制御に用いるレジスタなどが接続されている。また，ALUで行う算術／論理演算の前処理として，S2-Busのデータに対するシフト／マスク演算を施すことができる。この機能は，ALU演算結果に対するバイト・スワップ機能とともに，フィールド抽出／埋込みのために用いられる。

マイクロプログラムの順序制御機能としては，以下のものが設けられている。

(1) 無条件分岐:
絶対アドレスによる無条件分岐を行う。

(2) 二方向条件分岐:
ALU演算の結果を保持するフラグやスイッチ・フラグの集合体である，レジスタ **MSTR** の特定のビットのオン／オフにより，次の番地か，自己相対方式で指定された番地かのいずれかに分岐する。また，分岐条件として **WF** のアドレス・レジスタの値や，タグに関する比較結果によるものなども用意されている。これらについては，3.1.4, 3.1.5で述べる。

(3) 多方向分岐:
組込述語の呼出しに対応する，タグ `blt` が付されたデータを格納するレジスタ **IR** が用意されており，組込述語のコード（ビット31 ～ 24）をアドレスとした，多方向分岐テーブル **DPTM** の読出結果による分岐が可能である。また **DPTM** は，3.1.5で述べるタグによる多方向分岐にも用いられる。

(4) サブルーチン呼出／復帰:
マイクロプログラム・サブルーチンのために，1 Kwの深さを持つスタック **MSTK** が用意されており，マイクロ操作 `call`/`return` によって自動的にプッシュ／ポップがなされる。

(5) レジスタ間接分岐:
マイクロプログラム・アドレス・レジスタ **JR** の内容をアドレスとして分岐する。また，マイクロ命令で指定された絶対アドレスを **JR** にロードする機能もあり，アセンブラ／リンカによるアドレスの生成や管理に配慮した設計となっている。なお，**JR** はデクリメント機能と，その値が0であることによる分岐条件生成機能を持っており，ループ制御用カウンタとしても使用される。

これらの機能を用いて生成されたアドレスにより，64 ビット × 16 Kw の容量を持つ **WCS** がアクセスされ，その結果がマイクロ命令レジスタ **MIR** にセットされる。なお，**WCS** はマイクロプログラム自身によって書換えることができ，ダイナミック・ローディングやスクラッチ・パッドとしての使用が可能である。この他，実行したマイクロプログラムのアドレスをトレースする 1 Kw のリング・バッファである **TRCM** が備えられており，マイクロプログラムのデバッグ効率化に大きく寄与している。

以上述べたハードウェア回路は，汎用の MSI/SSI である FAST* [FAIRCHILD 82] 約 1,800 チップと，各種テーブルやレジスタ・ファイル類のための 190 チップの SRAM (4 ～ 16 Kbit/chip) を用いて構成され，200 ns のマシン・サイクルで動作している。また，これらの素子は約 30 cm 四方のプリント基板 12 枚の上に実装された。この他，主記憶は 256 Kbit の DRAM を用いて 16 枚（最大構成時）のプリント基板に，入出力制御装置は 10 枚のプリント基板にそれぞれ実装された。これらと周辺機器や電源などを含む筐体の大きさは，高さ 1,440 mm，幅 650 mm，奥行 900 mm となっている。

*FAST® は Fairchild 社の登録商標である。

3.1.4 レジスタ・ファイル

PSI-I のハードウェアの特徴の一つに，多目的に使用されるレジスタ・ファイルである **WF** の存在がある。WF は図 3-7に示すように，ハードウェア構成上は 4 つの領域に分けられ，また KL0 の処理系では 7 つの領域に分割して使用している。

まず先頭の 16 w は，S1/S2-Bus の双方に読出すことができ，かつマイクロ命令フィールドをアドレスとする直接アクセスが可能な領域である。従って最も使い勝手の良い領域であり，演算の途中結果の保持や，マイクロプログラム・サブルーチンの引数の受渡しに用いられる。

次の 48 w は，S1-Bus にのみ出力できる直接アクセス可能な領域である。この領域には，スタック・ポインタやスタック・フレームのベースなど，2 章で述べた様々なレジスタが配置されている。また，末尾の 64 w は読出に関してのみ直接アクセス可能な領域であり，

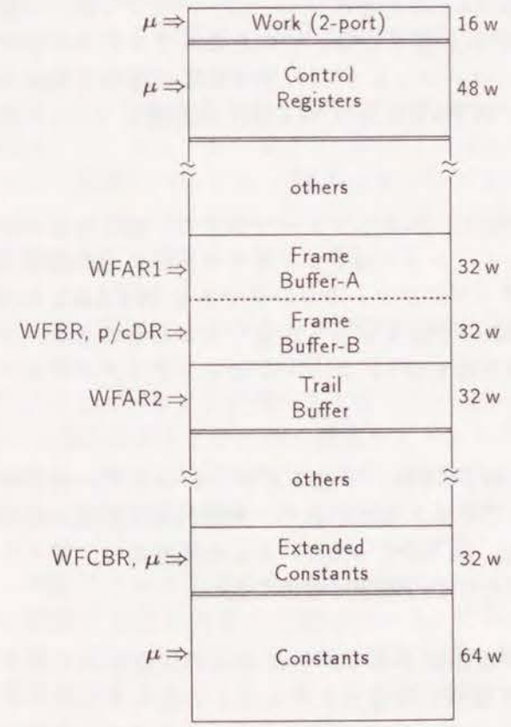


図 3-7: WF の構成

種々の定数を格納するために用いられる。

以上を除く領域は，様々なアドレス生成用レジスタを用いた間接アクセスのみが可能な領域であり*，マイクロプログラムにより4つの領域に分けて使用されている。中でも最も重要な領域が，3.1.2で述べたフレーム・バッファである。

フレーム・バッファのアクセスのために，2種類のアクセス・モードが用意されている。一つはアップダウン・カウンタであるレジスタ **WFAR1** を用いた順次アクセスである。このモードは，引数を一つずつ準備するための書込と，ヘッド・ユニフィケーションを一つずつ行うための読出の，双方に用いられる。また，Environment/Choice Point の生成のために，フレーム・バッファを主記憶に書き写す際にも，順次アクセス・モードが用いられる。

もう一つのアクセス・モードは，フレーム・バッファ中の局所変数をランダム・アクセスするために用いられる。局所変数の番号，即ちフレーム・バッファの先頭からの相対位置は，タグが `lvar/lref` であるワードの下位5ビットで与えられる。従って，メモリ・データ・レジスタである **pDR/cDR** の下位5ビットがWFのアドレスとして用いられる。一方，3.1.2で述べたようにFirst Goal Optimizationのためにフレーム・バッファは二つ設けられているため，どちらを使用するかの選択が必要となる。このため，ランダム・アクセス・モードでのアドレスの上位5ビットは，ベース・レジスタ **WFBR** が保持する値としている。従って，二つのバッファの切換は，**WFAR1** 及び **WFBR** の書換え（ビット反転）によって行うことができる。

間接アクセス領域にはフレーム・バッファの他に，スタック・バッファの一種であるトレイル・バッファも割付けられている。トレイル・バッファはトレイル・スタックの先頭部分（最大32w）をキャッシュしたものであり，アップダウン・カウンタである **WFAR2** を用いてプッシュ／ポップされる。また，**WFAR2** の下位5ビットが全て1であることが，マイクロプログラムの分岐条件の一つとして用意されており，オーバフロー／アンダーフローの検出に使用される。

また，アドレスの上位5ビットがレジスタ **WFCBR**，下位5ビットがマイクロ命令のフィールドによって生成される拡張ランダム・アクセス領域があり，比較的使用頻度が低い定数や制御レジスタが割付けられている。また，図3-7で'others'とした領域には，様々な制御情報やテーブル類が割付けられているが，その使用頻度は極めて低い。

以上のように **WF** には性質の異なった様々な領域があり，それぞれの用途に応じたアクセス・モードを実現するためのハードウェア機構が用意されている。これらが実際にどのように使用されているかを知ることは，PSI-Iのハードウェア・アーキテクチャを評価する上で重要な項目であるばかりではなく，その改良に関する重要な指針ともなる。そこ

*直接アクセス可能な領域も間接アクセスできる。

表3-1: WFのアクセス特性

Access Mode	S1-Bus	S2-Bus	DST-Bus
WF00-0f	12.2 / 6.9	100.0 / 29.1	33.0 / 12.1
WF10-3f	58.5 / 33.0	—	63.6 / 23.3
Constant	23.0 / 13.0	—	—
@pDR/cDR	1.3 / 0.8	—	0.3 / 0.1
@WFAR1	4.6 / 2.6	—	2.8 / 1.0
@WFAR2	0.1 / 0.0	—	0.3 / 0.1
@WFCBR	0.3 / 0.2	—	0.0 / 0.0
total	100.0 / 56.4	100.0 / 29.1	100.0 / 36.6

* x/y : x = WFアクセス命令中の頻度
y = 全命令中の頻度

で，SIMPOSのウィンドウ・システム（*Window*）と，構文解析システム *BUP* (Bottom-Up Parser) を対象に，**WF** の動的なアクセス特性を測定した[Nakashima 85, 87b]。

表3-1はその結果であり，**WF** がアクセスされる頻度をバスごとに集計した値と，その領域／アクセス・モードごとの内訳(S1-Bus/DST-Bus)が示されている。まず，全体のアクセス頻度については，S1-Busについては半数以上を占め，S2-Bus/DST-Busについても約1/3という，他のレジスタを大きく上回る値となっており，**WF** が重要な役割を果たしていることが明らかになった。

一方，領域ごとのアクセス頻度については，直接アクセス可能な領域に対するものが極めて多く，マイクロ命令の約1/3を占めるアドレスを指定のためのフィールドが，有効に利用されていることが明らかとなった。但し，各領域の半数のエントリに対するアクセスが95%以上であることから，領域やアドレスのビット数の削減が可能であることが示唆されている。特に定数領域については，有効ビットが少ない値の使用が支配的であり，マイクロ命令フィールドを直接バスに出力する方式*への変更も可能であることが判明した。

間接アクセス領域については，フレーム・バッファへのアクセスが主であるが，絶対的な頻度は予想を大きく下回っている。これは，ユニフィケーション以外の処理がかなり「重く」なっていること，実行頻度の高い組込述語の引数評価にフレーム・バッファが有効利用できなかったことなどが原因と考えられる[Nakajima 86a]。また，トレイル・バッファに関してはアクセス頻度が極めて低く，有効性が極めて疑わしい。実際PSI-Iのハードウェアでは，この方式を廃止した場合の性能低下は極微であることが明らかになっており

*S2-Busへの定数出力はこの方式である。

[Nakajima 86b]，方式自体の有効性を再検討する必要があるという結論に達した。

なお，**WF** 以外のレジスタについては，メモリ・インタフェースである **pLAR/cLAR** と **pDR/cDR** のアクセス頻度が高く，これらを重視した方針が正しかったことが明かとなった。また，定数生成の頻度がかなり高いことも，見逃すことのできない特性である。

3.1.5 タグ・アーキテクチャ

本章の冒頭でも述べたように，タグ・アーキテクチャは PSI 系の推論マシンにすべて共通した重要な特徴であり，PSI-I においてもタグ操作のための様々なハードウェア機構が用意されている。これらの機構は，タグとアドレス／数値などの「値」をハードウェア・レベルで並列に処理するために設けられている。例えば；

```
Z.tag = X.tag ;
Z.value = X.value + Y.value ;
if (Y.tag == int) ...
```

のように，タグと値を分離して演算を行い，同時にタグを判定する操作は，極めて頻繁に行われる。PSI-I では，これらの操作を並列に行うことができるように，演算系回路においてタグと値を分離するとともに，それと並行動作する順序制御系回路に様々なタグ判定機構を用意している。

3.1.5.1 演算系

図 3-8に示すように，メモリ・データ・レジスタである **pDR/cDR** と，レジスタ・ファイル **WF** にタグが付加されている。従って，これらのレジスタが接続されている S1-Bus/S2-Bus/DST-Bus は，全てタグを持っている。

タグに関する演算／転送の基本機能は，図 3-8に実線で示したものである。即ち，DST-

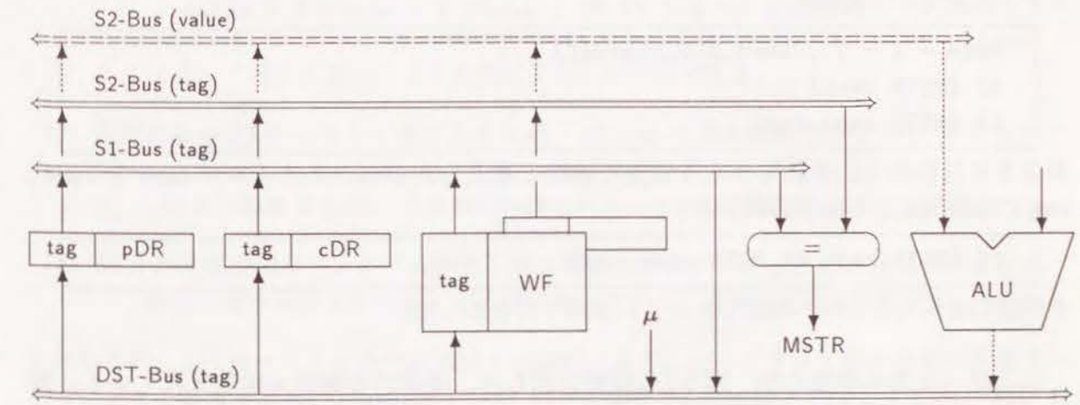


図 3-8: PSI-I 演算系のタグ・アーキテクチャ

Bus のタグ部に出力される値は；

- S1-Bus のタグ部
- マイクロ命令で指定された即値 (μ)

のいずれかである。後者は，特定のタイプのデータを生成するのに用いられる。例えば，二つの未定義変数の X, Y のユニフィケーションにおいて，X へ Y への Reference Pointer を代入する際に；

```
pLAR = "address of X" ;
pDR.tag = ref , pDR.value = "address of Y" ;
write(pLAR,pDR) ;
```

というように，ref が即値として指定される（上記の2行目のような ',' による複数の操作を連結は，これらの操作が1マシン・サイクルの中で並行して実行されることを意味する。また ';' はマシン・サイクルの切れ目を意味する。以下同様）。この機能はかなり重要なものと考えられるが，使用頻度を測定したところ，前述の *BUP* で 2%，*Window* では 3% と予想を大きく下回る値であった。この理由は，プログラム中に出現する定数や構造体が，KL0 の内部表現にそのまま出現している（即ちタグが既に付いている）ことによるものと思われる。また 3.1.7 で述べるように，タグ即値生成操作がマイクロ命令の構成上，他の操作と並行して行いにくいため，後述する別の生成操作を用いていることも大きな原因の一つと考えられる。

さて，このタグ転送／生成と操作とは並行して，S1-Bus と S2-Bus のタグ部の一致比較演算が行われる。この比較結果は，ALU 演算のステータス（符号，ゼロなど）とともにステータス・レジスタ **MSTR** にセットすることができる。例えば，二つのレジスタ X と Y のタグ／値双方の一致判定は；

```
void = X - Y , load_ALU_status() ;
if (MSTR.zero) ...
if (MSTR.same_tag) ...
```

のように行われる。これもかなり重要な機能と考えられるが，ステータス zero と same_tag の論理積による条件分岐，即ち；

```
if (MSTR.zero && MSTR.same_tag) ...
```

を用意しなかったため，期待したほどは使用されなかった。

以上述べた基本機能の他，図 3-8に破線で示した，補助的な機能も用意されている。即ち，S2-Bus のタグ部を ALU の入力とする機能と，ALU 演算の結果を DST-Bus のタグ部に出力する機能である。これらはいずれもタグを「値」として用いる「異例の」機能であり*，その使用頻度は小さいものと予測された。しかし，前述のタグ即値生成操作や，後述

*必須ではある。

するタグと即値の判定操作が，マイクロ命令の構成の関係で使用しにくいことから，例えば *BUP* ではタグを ALU 入力とする操作が 2.5% ほど出現している（*Window* では 0.4%）。しかし，この結果からこれらの機能を重要視するのは短絡的であり，むしろ即値に関する操作の充実を図るのが正しい方向であると判断される。

3.1.5.2 順序制御系

順序制御系におけるタグ操作は，図 3-9に示すように，二方向分岐の条件生成と，テーブルを用いた多方向分岐の二つに大別される。

タグに関する分岐条件の中で最も重要なものが，S2-Bus のタグと即値の比較である。この機能は，組込述語でのデータ型のチェック，デレファレンス，また一方のデータ型が既知であるようなユニフィケーションなど，様々な局面で使用されると予測された。しかし，使用頻度を測定した結果では，*BUP* で 2.7%，*Window* で 5.2% であり，さほど大きな値ではなかった [Nakashima 87b]。特に，他の条件分岐が約 35%，タグによる多方向分岐が約 10% の使用頻度であることを勘案すると，このような低頻度となった背景には，何らかのアーキテクチャ上の問題点が潜んでいると考えられる。

そこで，様々な見地から解析を行った結果，以下のような問題点が明らかとなった。

(1) 比較結果が「等しい」時に分岐する操作のみが用意されているため，組込述語のタイプ判定，即ち；

```
Z.value = X.value + Y.value , if (Y.tag != int) ...
```

のように，「等しくない」ことを判定する際に不便である。

(2) 比較対象が S2-Bus のタグ部であるため，S1-Bus にのみ接続されている **WF** 上のフレーム・バッファ／トレイル・バッファなどのデータに，直接適用できない。その結果，**pDR/cDR** に転送して多方向分岐を行うパターンが増加している。

(3) KL0 の内部表現がプログラム中の定数／構造体を「データ」として扱っているため，「一般的」なユニフィケーション処理になりやすく，多方向分岐が多用される。

これらの内，(1) はマイクロ命令の構成上の問題であり，ハードウェア量や速度に影響を与えることなく解決できる。(2) については，判定対象を S1-Bus のタグ部とすると，**WF** の全体と先頭の 16 w とのアクセス速度の差（異なる素子を用いて実現している）や，アドレス生成に要する時間の差が問題となり，サイクル・タイムに影響を及ぼす可能性がある。しかし，ハードウェア量はほとんど変わらず，またマイクロプログラムのステップ数削減に対する効果が大きいことが期待できるため，魅力的な改良項目であると考えられる。(3) は

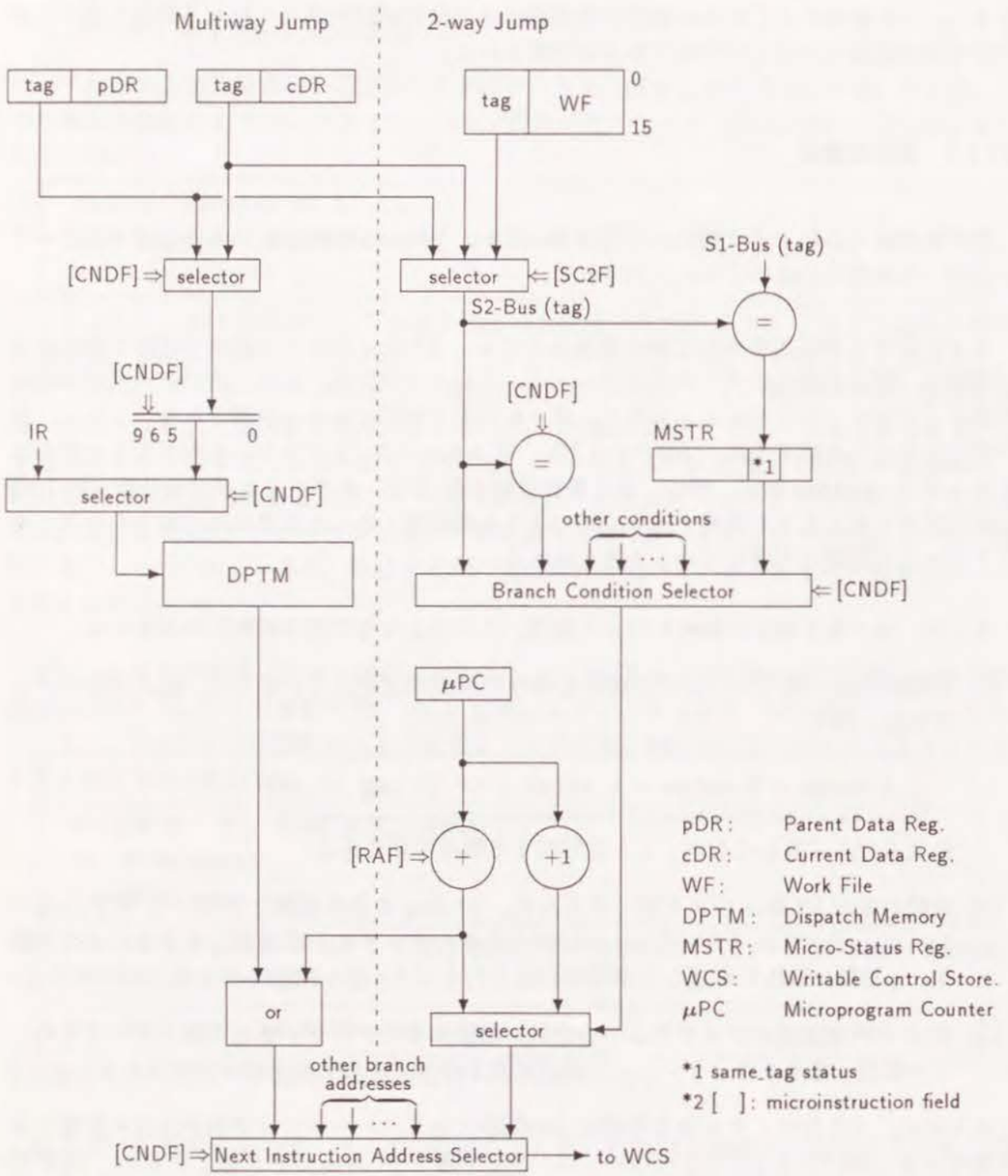


図 3-9: PSI-I 順序制御系のタグ・アーキテクチャ

ハードウェアには直接関係しない問題点であるが，処理方式を変更すれば使用頻度が増加し，ハードウェアやマイクロ命令の構成にも影響する可能性があることが判明した。

この他の分岐条件として，前述の **MSTR** の **same_tag** ステータスと，S2-Bus のタグ部の各ビットのオン／オフが用意されている。後者は主に，ガベージ・コレクタによる GC ビット（タグの上位 2 ビット）の判定のために用意したが，他の用途もあることが判明した。即ち，タグを対象とする演算や多方向分岐によって，いくつかのデータ型からなる集合に属することを判定し，その集合に共通した処理を行った後，ビットのオン／オフで個々のデータ型を切り分ける手法が，しばしば用いられていた。このことから，例えばマスク付きの比較のような，データ型の集合の判別操作も，タグ判定操作の一つとして考えられることが判った。

タグを用いた多方向分岐に関しては，マイクロプログラムの容量や書きやすさに配慮して，柔軟性の高い構成とした。まず，多方向分岐の対象となるのは **pDR/cDR** のタグであるが，これを直接に分岐アドレスとはせず，RAM テーブルである **DPTM** を介したアドレス生成方式を採用した。また，**DPTM** を 12 の「バンク」に分け，**pDR/cDR** のタグをバンク内アドレスとし，バンクの選択はマイクロ命令フィールドで指定する構成とした。更に，**DPTM** の各エントリには分岐アドレスそのものではなく，ベース・アドレス（自己相対）に OR されるオフセットを格納することとした。

まず **DPTM** による間接アドレス生成は，異なる型のデータに対して同じ処理を施すのに適している。例えば X と Y のユニフィケーションおいて，X が整数であることが明らかになった後の処理は；

```
switch (Y.tag) {
    case 'ref' :   "continue dereference" ;
    case 'undef' : "store X in Y" ;
    case 'hook' :  "store X in Y and invoke hooked predicate" ;
    case 'int' :   "compare X and Y" ;
    default :      "backtrack" ;
}
```

のようなものとなる。この例における **default** に対する処理は，**DPTM** の対応する各々のエントリに同じ分岐オフセットを格納しておくことにより，同じマイクロプログラムにより実行することができ，マイクロプログラム容量が大幅に節約できる。また，各々の **case** に割当てられる WCS の連続した領域を，処理の内容に応じて調整することもできる。

次に，**DPTM** を複数のバンクに分けたことにより，処理の内容に応じて異なったタイプの多方向分岐が可能となっている。例えば Undo 処理における多方向分岐は；

```
switch (*--TR) {
    case 'undef' : "make unbound" ;
    case 'save' :  "restore saved value" ;
    case 'excb' :  "restore EXTR" ;
    case 'obcb' :  "invoke on-backtrack predicate" ;
    default :      "fatal error" ;
}
```

のようなものである。これは，前述の整数に対するユニフィケーションに関するものとは全く異なるものであり，DPTMの異なるバンクが使用される。

一方，DPTMが保持する値をオフセット値としたことにより，類似した多方向分岐を同じバンクを用いて実現することができる。例えば前述の整数に関するユニフィケーションを；

```
switch (Y.tag) {
    case 'ref' :   "continue dereference" ;
    case 'undef' : "store X in Y" ;
    case 'hook' :  "store X in Y and invoke hooked predicate" ;
    case 'int' :   "compare X and Y" ;
    case 'atom' :  "backtrack" ;
    default :      "backtrack" ;
}
```

とすると，アトムに関するユニフィケーション処理；

```
switch (Y.tag) {
    case 'ref' :   "continue dereference" ;
    case 'undef' : "store X in Y" ;
    case 'hook' :  "store X in Y and invoke hooked predicate" ;
    case 'int' :   "backtrack" ;
    case 'atom' :  "compare X and Y" ;
    default :      "backtrack" ;
}
```

と同じ分岐パターンとなる。従って，これらを同じバンクに割当ててバンクを節約する一方で，ベース・アドレスを固有のものとすることによって個別の処理を実現することができる。

以上のようにタグによる多方向分岐のためのハードウェアは，極めて柔軟性の高い構成となっている。このようにマイクロプログラマにとって使いやすい機能とした効果が，*BUP*では10.9%，*Window*では8.6%という高い使用頻度に現れている。

なおDPTMの末尾256wは，組込述語を処理するマイクロプログラム・ルーチンへの分岐に用いられ，IRの上位8ビット（組込述語コード）によりアクセスされる。また，組込述語の引数情報はIRの下位24ビットに，8ビットずつ設定され，各バイトの上位3ビットが引数のタイプ（局所変数，小さい整数など）を示す。そこで，IRの各バイトの上位3ビットを分岐アドレスのビット3～1に埋め込む多方向分岐も用意されている。

3.1.6 メモリ・アーキテクチャ

3.1.6.1 アドレス変換機構

PSI-Iのメモリ・アーキテクチャの最大の特徴は，複数スタックの効率的管理のための2レベルアドレス変換機構である。PSI-Iではローカル・スタック*，コントロール・スタック*，グローバル・スタック，及びトレイル・スタックの，4つのスタックが使用される。これらはプロセスに固有，即ち個々のプロセスに対して独立に割当てられる領域である。一方，ESPのオブジェクトやコードが割付けられるヒープは，コードの共有やプロセス間通信の実現のために，全てのプロセスから共有される領域となっている。

さて，汎用機上の処理系などでは，複数のスタックが一つの連続した記憶空間に割当てられる。従って，記憶空間が尽きる前にスタックが衝突する可能性があり，その際にはスタックの移動(Stack Shift)という極めて手間のかかる処理が必要となる。

そこで，固有／共有の領域管理と，独立に伸縮するスタックの管理のために，PSI-Iには図3-10に示すように，2レベルのアドレス変換機構が備えられている。まず，論理アドレ

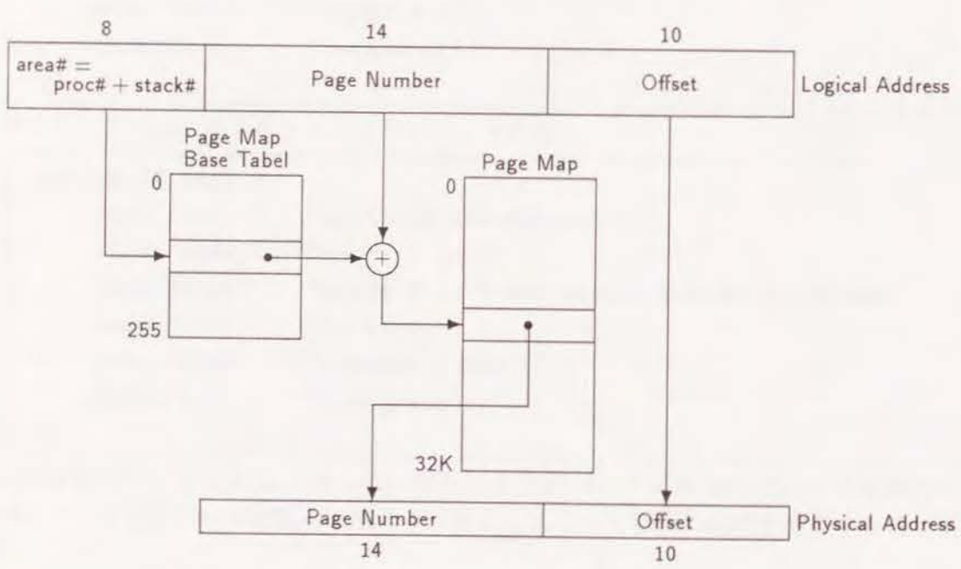


図 3-10: PSI-Iのアドレス変換機構

*Environment/Choice Point中の，引数及び局所変数がローカル・スタックに，それ以外の情報がコントロール・スタックに割当てられる。

スの上位8ビットである「エリア番号」は，プロセスのIDとスタックのIDを組合せたものとなっている。即ち，上位6ビットがプロセス番号を表し，下位2ビットがスタックの種別を表す。但し，上位6ビットが0であるようなエリアは共有領域であり，ヒープやマイクロプログラムが使用するシステム領域となっている。このような構成とすることにより，各々のスタックに対して16 Mwの論理空間が与えられ，大規模なプログラムにも充分に対応することができる。

さて，論理アドレスから物理アドレスへの変換は，ページマップの索引により行われるが，その前段階としてエリア番号からページマップのベースへの変換が行われる。即ち，エリア番号をアドレスとしてページマップ・ベース・テーブルが索引され，その結果得られた値に論理ページ番号（論理アドレスのビット23 ～ 10）を加算した結果が，ページマップのアドレスとなる。この方式によって，ページマップの容量を比較的小さなものとすることができ，SRAMを用いたアドレス変換機構のハードウェア化が可能となった。なお，ページマップは物理記憶空間（16 Kページ）の2倍の容量があり，二つのエリアがページマップ上で衝突する頻度を極めて小さなものとしている。

物理ページの割当ては，マイクロプログラム化されたデマンド・ページングによって行われる。即ち，スタックやヒープが伸長して，物理ページが割当てられた領域を超過しようとした時，マイクロプログラムによって未使用の物理ページが即座に獲得される。但しその際に，ページマップ上での衝突や，物理ページの払底が検出されると，OSの記憶管理ルーチンに割り出される。記憶管理ルーチンは，スタックの縮退によって使用されていないページの解放や，ページマップの移動*を行う。更に，この処理によっても充分な物理ページが獲得できない場合には，ガベージ・コレクタが起動される。

なお，デマンド・ページングのためにはスタック伸長時に割当ての有無を知る必要があるが，この判定は簡単なハードウェア機構を用いて行われる。即ち，マイクロプログラム中でスタック・トップを増加するための加算をALUを用いて行う際，ページ境界を超えたことを意味するビット9からのキャリーが**MSTR**にセットされる。マイクロプログラムではこのフラグのオン／オフを判定し，ページ境界を超えた時にのみスタック・トップと割当済み領域の末尾との比較を行う。この機構は最小限のハードウェアで実現され，また常に比較演算を行う汎用機上の処理系などでの方式よりも，かなり優れたものである。しかし，ビット9からのキャリーの判定は常に必要であり，またページ境界付近でスタックが伸縮した場合には，割当の有無の判定も頻繁に行われる。従って，より高度なハードウェア・サポートの導入を含めた，改良の検討が必要であるという結論に達した。

*Stack Shiftに比べて極めて軽い処理である。

3.1.6.2 キャッシュ・メモリ

前述のようにキャッシュ・メモリの構成は，8Kwの容量，2ウェイのセット・アソシアティブ方式という，かなり大規模な構成とした。また，ライン・サイズは4wであり，書込の方式としてStore Throughよりも一般に高性能とされているStore Back方式を採用した。これらに関する評価は5.3.1で詳しく述べるが，95%以上のヒット率という充分に高い性能を達成することができた。

なお，キャッシュ・メモリの存在とスタックの特性を意識した，*write-stack*という特殊な書込操作を導入した。Store Back方式では，書込操作でキャッシュ・ミスが起こった際にも，キャッシュ・ラインに含まれる他のワードを得るために，物理記憶からキャッシュへのブロック・ロードが行われる。しかし，スタックの伸長を伴う書込の際には，スタック・トップよりも大きいアドレスのデータは無意味であり，それが参照されることはありえない。そこで，write-stack操作ではキャッシュ・ミスの際にもブロック・ロードを行わないこととした。従って，キャッシュ・ミス時の性能は通常の書込よりもかなり高く，特にストア・バックを伴わない場合にはキャッシュ・ヒットと全く同じ性能が得られる。

例えば，A～Eの5つの要素からなるスタック・フレームを，スタック・トップに生成する操作は，write-stackを用いて以下のように実現される。

```
write(SP++, A) ;
write_stack(SP++, B) ;
write_stack(SP++, C) ;
write_stack(SP++, D) ;
write_stack(SP++, E) ;
```

即ち，最初の要素Aに関しては，SPがキャッシュ・ラインの先頭にはない可能性があるので通常の書込操作が必要であるが，他の要素についてはwrite-stackを適用できる。例えば要素Bがキャッシュ・ラインの先頭であればブロック・ロードは不要であり，また先頭でなければAの書込によって必ずヒットすることが保証される。

なお，スタック・フレームの最初の要素に対しては通常の書込が必要であるという制限は，マイクロプログラムのループを記述する場合に不便である。また，1ワードをプッシュする操作にはwrite-stackを使用できないという欠点もあり，改良の余地があると考えられる。

キャッシュ・ミスの際の同期については，ミスが発生した時にマイクロ命令の実行を直ちに停止するのではなく，真に同期が必要になった時点で停止する方式を採用した。例えば，読出操作でミスした際には；

- 読出データを使った操作の実行

- 次のメモリ・アクセス

のいずれかが発生するまでは，マイクロ命令の実行は停止しない。この方式は，キャッシュとプロセッサの並行動作を可能としており，特にランダムな書込の際に効果があると予想される。実際5.3.1で述べるように，キャッシュ・ミスのペナルティを半分以下にすることができた。この他，読出でのキャッシュ・ミスの際には，ブロック・ロードをアクセス対象となったワードから開始し，かつそのデータが物理記憶から到着した時点でマイクロ命令実行を再開させるLoad Through方式や，ブロック・ロードをストア・バックに先行して行う方式の採用など，キャッシュ・ミス・ペナルティを最小化する工夫がなされている[Pohm 83]。

3.1.7 マイクロ命令アーキテクチャ

水平型のマイクロ命令のメリットは，ハードウェアの制御を並列に行うことにより，基本的に並列動作可能なハードウェアの潜在能力を充分に発揮させることにある。しかし，ハードウェア・レベルの潜在的な並列性は極めて高く，これを完全に並列制御するためには膨大なビット数のマイクロ命令を必要とする。従って，何らかのエンコードを行ってマイクロ命令を許容できるビット数としなければならないが，そのエンコード方法の良否，即ち並列制御するハードウェアの選択の良否が，性能を大きく左右することは明らかである。

PSI-Iのマイクロ命令は，図3-11に示すように63ビットであり，13のフィールドからなる43ビットの基本部分と，フィールド**TYPF**によりその解釈が定まる（タイプ1～3）20ビットの可変部分からなる。以下，各フィールドの機能について概説するとともに，*BUP*と*Window*における測定結果に基くマイクロ命令アーキテクチャの評価を行う[Nakashima 87b]。

(1) DBGF

DBGFはマイクロプログラムのデバッグや評価のためのフィールドであり，ブレーク・ポイントの設定，または評価用カウンタ**GEVC1/2**のインクリメントを指定することができる。このフィールドを設けたことにより，マイクロプログラムの論理を変更することなくデバッグ／評価を行うことができ，その有効性が実証された。しかし，デバッグと評価を同時に行うことは考えにくく，DBGFを1ビットとして具体的にどのような操作を行うかをフラグなどで定める方法も考えられる。

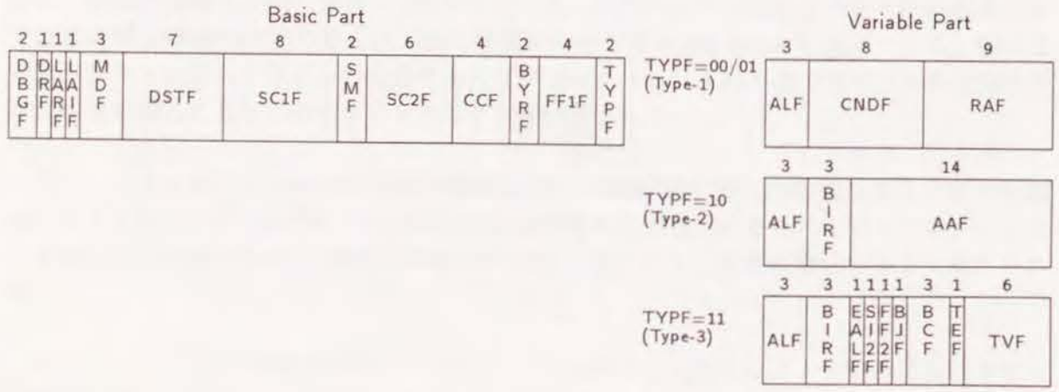


図 3-11: PSI-Iのマイクロ命令

(2) DRF/LARF/LAIF/CCF

これらのフィールドはメモリ・アクセスのために用いられる。即ち，**DRF**がデータ・レジスタ**pDR/cDR**の選択を，**LARF**がアドレス・レジスタ**pLAR/cLAR**の選択をそれぞれ行い，具体的な操作は**CCF**が定める。また，**LAIF**は選択されたアドレス・レジスタのインクリメントの有無を指定する。従って，データ／アドレス・レジスタの任意の組合せが可能である。

測定結果によれば，二つのレジスタのどちらが用いられるかは，アドレス／データともほぼ拮抗しており，二つのレジスタを設けた効果が証明された。但し，約90%が**pLAR/pDR**または**cLAR/cDR**の組合せであり，ビット数の削減の可能性が示唆されている。また，**CCF**についても，キャッシュ・パージなどの極めて使用頻度が低い操作が含まれていることから，単独または**DRF/LARF**と組合わせたビット数削減が可能と考えらる。

一方，**LAIF**によるアドレス・レジスタのインクリメントについては，メモリ・アクセスを行ったサイクルの約半分でこの機能が使われており，その内の90%以上が同時にALU演算を行っていることから，その有効性が実証された。

(3) MDF/DSTF

これらのフィールドは，DST-Busの値を書込むレジスタの選択に用いられる。**MDF**は**DSTF**を修飾するフィールドであり，その値が0～5の場合には**DSTF**は**WF**のアドレス生成に，また6～7の場合には個別のレジスタの指定にそれぞれ用いられる。但し，**MDF**の値によって，**WF**と**pLAR/cLAR/pDR/cDR**のいずれかへの同時書込や，**JR**のデクリメントが指定できる。

この同時書込機能や**JR**のデクリメント機能は，いずれも10%程度使用されており，有効であったことが確認された。その反面3.1.4で述べたように**WF**の直接アドレス領域の削減が可能であることや，no-operationのコードが多数あることから，**MDF**と**DSTF**を合わせた10ビットは，かなり削減できると考えらる。

(4) SC1F/SC2F

これらのフィールドは，S1-Bus/S2-Busに出力するレジスタを選択する。**SC1F**については3.1.4で述べたように，**WF**の直接アドレス領域の縮小によるビット数削減が可能であると判断できる。また，**SC2F**については**WF**の先頭16 wのタグ部をS2-Busに出力するコードなど，約半数のコードがほとんど使用されておらず，最低1ビットは削減可能と考えられる。

この他，S2-Busへの定数出力が10%以上あることが判明し，**WF**の定数領域のアクセ

スも同程度であることから，定数生成の重要性が再認識された。また，ハードウェア構成上の問題ではあるが，**WF** の間接アクセス領域（特にフレーム・バッファ）やポインタ類が格納されている直接アクセス領域を，S2-Bus に出力できないことに対する不満が，マイクロプログラマの多くから指摘された。

(5) SMF/BYRF/BIRF

これらのフィールドは S2-Bus に対するシフト／マスク演算を指定する。即ち，まず最初に **BYRF** で指定されたバイト数に **BIRF** で指定されたビット数を加えた値のローテート・シフトが行われる。その結果に **SMF** で指定される下位 16, 8, 5 ビットのいずれかを抽出するマスク演算が行われて，ALU に入力される。

マスク演算の使用頻度は約 15%，バイト・シフトの使用頻度は約 10% と，まずまずの値であるが，合計 4 ビットを投資した割には物足りない数値である。また，これらの演算は主に組込述語の引数情報の抽出に用いられており，例えば **IR** の特定のバイトをオフセットとする **WF** のフレーム・バッファのアクセスなど，その操作自体に適した別のハードウェアを用意する方が効果的であると考えられる。

ビット・シフトに関しては使用頻度が 3% 以下であり，マイクロ命令フィールドの投資効果だけではなく，ALU と直列に配置したハードウェア構成自体が疑問視される。また一方では，ビット数指定がマイクロ命令に埋め込まれているため，例えばシフト演算を行う組込述語のように，ビット数がデータとして与えられる処理が実現しにくいという指摘もあった。

(6) FF1F/FF2F

これらのフィールドは **MSTR** の制御を行う。**MSTR** は ALU 演算の結果を保持する 10 ビットのフラグ（ALU フラグ），個別にセット／リセットが可能な 8 ビットのフラグ（スイッチ・フラグ，**SW0** ～ **7**），及び機械命令レベルでのコンディション・コードとしての使用などを想定した 14 ビットのフラグからなる。**FF1F** では ALU フラグの一括設定，符号またはゼロ・フラグのみの設定，または **SW0** ～ **3** に対する個別のセット／リセットが指定できる。**FF2F** はタイプ 3 のマイクロ命令にのみ設けられており，オンである時には **FF1F** の意味が変更され，**SW4** ～ **7** に対する個別のセット／リセットなどが指定できる。

まず，ALU フラグについては，その設定操作が 15 ～ 20% の頻度で行われており，比較的高頻度な操作であると言える。その中での一括設定と符号／ゼロ・フラグのみの設定の比率は約 2:3 であったが，マイクロプログラマの大多数によれば一括設定のみで充分であるとの意見であった。また，アドレス演算を意識して 24 ビット演算系のフラグを用意したが，これらはほとんど無意味であることが明らかになった。

一方スイッチ・フラグに関しては約 7% の頻度で操作されており，無視できない値であると言える。また，タイプ 3 のみで可能な **SW4** ～ **7** の操作は約 2% であったが，タイプ 3 のマイクロ命令の中では 30 ～ 40% の頻度となっている。従って，分岐がほとんど不可能な「不便な」タイプであるタイプ 3 が選択された理由のかなりの部分が，この操作にあると言うことができ，注目に値する結果となっている。

なおその他のフラグに関しては，ほとんど操作されることがなく，その存在自体が疑問視される。

(7) TYPF

TYPF は，マイクロ命令の下位 20 ビットの解釈を定めるフィールドであり，0 または 1 のがタイプ 1，2 がタイプ 2，3 がタイプ 3 をそれぞれ意味する。なお，**TYPF** が 1 の時と，タイプ 3 で **SI2F** が 1 の時に，**SC2F** の値が定数として S2-Bus に出力される。

比較的使用頻度が高い操作に関して，タイプごとに実行可能なものをまとめると，以下のようになる。

タイプ 1: 算術演算，S2-Bus への定数出力，絶対アドレス以外での分岐
タイプ 2: 論理演算，絶対アドレス (**AAF**) での分岐
タイプ 3: 算術／論理演算，S2-Bus への定数出力，**SW4** ～ **7** の操作，タグ即値出力

さて，各タイプの使用頻度は，タイプ 1 が約 65%，タイプ 2 が約 10%，タイプ 3 が約 7%，タイプを問わないものが約 18% であった。従って，分岐の頻度はタイプ 1 とタイプ 2 を加えた約 75% となり，極めて高い数値であることが判明した。逆に言えば分岐がほとんど不可能なタイプ 3 でのみ可能な **SW4** ～ **7** の操作やタグ即値出力は，極めて使用しにくいものであると言える。従って，これらの機能を分岐と同時に行えるような工夫が必要であると考えられる。

(8) CNDF

CNDF は分岐の種類と分岐条件を指定するフィールドであり，前述のように約 65% の頻度で使用されている。その内訳は，二方向条件分岐が約 40%，タグによる多方向分岐が約 10%，その他の無条件分岐（サブルーチン・コールや復帰などを含む）が約 15% であり，二方向条件分岐の頻度が高いことが際だった特徴となっている。但し，3.1.5 で述べたようにタグに関する条件分岐に不備があることや，**MSTR** のかなりのビットが削減できることなど，改善の余地は多いと考えられる。また，組込述語への分岐やサブルーチンからの復帰など，自己相対分岐アドレスである **RAF** を必要としない操作が約 7% あったことや，

比較的近傍への分岐が支配的であることから，アドレスの指定方式の改良も考慮する必要がある。

(9) ALF

ALF は ALU で行う演算の種類を定めるフィールドであり，タイプ1では算術演算が，タイプ2では論理演算が，またタイプ3では **EALF** により算術／論理演算の双方が，それぞれ指定できる。

ALF に関する評価結果では，まず全くデータ転送や演算を行わないものが約20%あったことに注目する必要がある。この結果と分岐頻度の高さを考え合わせると，分岐のみを行っているマイクロ命令がかなり存在することが結論される。実際，マイクロプログラムの最適コーディングを行おうとすると，データ転送／演算の数よりも条件分岐の数によってステップ数が定まることがしばしばある。これを改善するには，前述の `same_tag` と `zero` の論理積のような，複合条件による条件分岐の導入が必要と考えられる。また，`Y = X + 0` のような単なるデータ転送が約40%あることも，注目すべき結果である。

残りの40%では何らかの演算が行われ，その内の約80%が加減算であった。なお，アドレス計算用に24ビットの加減算をサポートしているが，前述のようにほとんど無意味であることが明らかになっている。また約20%を占める論理演算の大半がAND演算であったが，これは **SMF** によるマスク演算が適用できないフィールド抽出や，変数がローカル／グローバルのどちらのスタックにあるかの判定のために用いられていると思われる。この他，ALU演算の結果に対するバイト・スワップは全くといって良いほど使用されず，回路自体の存在が疑問視される。

(10) BJF/BCF/TEF

これらのフィールドはタイプ3にのみ存在する。**BJF** は **JR** をアドレスとする間接分岐を，**BCF** はI/Oバスの制御を行うが，いずれも1%以下の使用頻度であり，マイクロ命令フィールドとしての存在価値は極めて疑わしい。

TEF はタグ即値の出力を制御し，オンである時に **TVF** がDST-Busのタグ部に出力される。前述のように，タグ即値の使用頻度は2%程度であったが，これはタイプ3でのみ実行可能な操作であることに起因していると思われる。実際，タイプ3のマイクロ命令に関しては，30～40%の使用頻度であり，タイプ3が選択された主要因の一つとなっている。

3.2 コンパイル方式による逐次型推論マシン (PSI-II)

3.2.1 設計方針

前節で述べたようにPSI-Iの設計においては，ハードウェアと処理方式の両面において，推論マシンのアーキテクチャを模索している段階であった。このため，マイクロ・インタプリタ方式と言う柔軟性の高い処理方式を採用したが，方式上の問題点，即ち大キなセマンティクス・ギャップを埋めるためのマイクロプログラム処理のオーバヘッドにより，必ずしも満足できる性能を達成することができなかった [Nakajima 86a, Nakashima 87b]。またハードウェアについても，前節で述べたような機能の過不足が様々な点で指摘され，これらの改良が処理速度とハードウェア量削減の双方に大きく寄与することが明らかになった。

一方，第五世代コンピュータ・プロジェクトが進行するに従って，並列推論に関する研究が次第に活発化し，KL1を用いた様々な並列ソフトウェアが開発され始めた。しかし，このような研究を本格化するには，その当時使用可能であった逐次的な処理系や6台のPSI-Iを結合したMulti-PSI/v1 [Masuda 88] は，性能と処理可能なプログラム規模の両面で貧弱なものであった。そこで，プロセッサ数，単体性能，通信速度，メモリ量，実用性など，あらゆる面で本格的な並列推論マシンの実現が強く望まれていた。

PSI-II [Nakashima 87c] はこのような背景に基き，PSI-Iの後継機として更に高度なAIワークステーションを提供することと，本格的並列推論マシンMulti-PSI/v2 [Takeda 88, Uchida 88] の要素プロセッサとすることの，二つの目的のために開発された。これらの目的を達成するために，PSI-IIでは高速化と小型化を特に重視する方針で設計を行った。

高速化の面では，まずマイクロ・インタプリタ方式を改め，[Warren 83] に基く機械命令セットをエミュレーションする方式とし，セマンティクス・ギャップの圧縮によるオーバヘッドの削減を図った。この結果，PSI-Iではマイクロプログラムが行っていた操作がコンパイラに委譲され，プログラムが持つ様々な情報が低レベルの処理系に解釈しやすい形で提示されることとなり，頻度の高い比較的単純な操作の高速化に大きな効果があった。また，2.4で述べた基本的な最適化手法のより効率的な実現や，4章で述べる新しい最適化手法の導入も，高速化に大きく寄与している。

ハードウェアについては，タグ，記憶管理，マイクロプログラミングの三大基本方針を踏襲しつつ，インタプリタからエミュレータへと一新された処理方式の変更にマッチした機構の導入を中心に，大幅な改造を行った。また，前節で述べたハードウェア機構やマイクロ命令アーキテクチャの不備を補い，投資したハードウェアの効果を最大限に発揮できるような構成とした。

これらの結果，PSI-Iの3～11倍という極めて大幅な性能向上が実現されたばかりではなく，430 KLIPSという当時では世界最高の性能を達成することができた。

一方，小型化の面ではCMOSゲートアレイを始めとするLSIの採用が最も大きく寄与したが，ゲート搭載量やピン数など様々な制限を持つLSIを効果的に使用するための工夫も，ハードウェア構成の上で見逃すことができない。またPSI-Iでの評価に基く，実装上の問題となるような機構の圧縮や削除も，小型化に大きく貢献した。その結果，CPUの実装規模はPSI-Iの1/4に圧縮され，64プロセッサという大規模な並列推論マシンの実現が可能になった。また，主記憶や入出力機器の実装規模も削減された結果，筐体容積はPSI-Iの約1/6となり，コンパクトなAIワークステーションを提供することができた。

3.2.2 機械命令アーキテクチャ

3.2.2.1 WAM

PSI-IIの機械命令は，[Warren 83]で提案された *Warren Abstract Machine (WAM)* と呼ばれる抽象マシンの命令セットを基本としている。WAMの記憶空間やレジスタについては2章で述べたが，具体的な命令の意味とともに，PSI-IIでの実現方式に則して再述する。

まず，記憶空間は；

- ローカル・スタック
- グローバル・スタック
- トレイル・スタック

の三つのスタックと*，KL0特有の領域であるヒープに分割される。なお，[Warren 83]では構造体のユニフィケーションのためのスタック(PDL)が用意されているが，PSI-IIではローカル・スタックを用いるため不要である。

各スタックのトップやスタック上のフレームへのポインタなど，制御用のレジスタとしては以下のものが存在する。

P	プログラム・カウンタ
CP	復帰アドレス
E	最新のEnvironmentのベース
B	最新のChoice Pointのベース
L	ローカル・スタック・トップ†
G	グローバル・スタック・トップ†
GB	グローバル・スタックのバックトラック・ポイント†
TR	トレイル・スタック・トップ
S	構造体ユニフィケーションのためのポインタ

またKL0特有の遠隔カットのために，レベル・カウンタ **LVLC** が用意されている。

述語を呼出す際にはその n 個の引数が，レジスタ $\mathbf{A_1} \sim \mathbf{A_n}$ に格納される。なお，引数レジスタ $\mathbf{A_i}$ は一時変数レジスタ $\mathbf{X_i}$ としても使用される。一時変数レジスタは，ヘッドと第一ゴールにのみ出現する変数，最終ゴールにのみ出現する変数，及びヘッド・ユニフィケーションやゴール引数生成時の副構造体の格納のために用いられる。

*[Warren 83]ではそれぞれStack, Heap, Trailと呼ばれる。
†[Warren 83]では **L**, **G**, **GB** はそれぞれ **A**, **H**, **HB** と呼ばれている。

表 3-2: *get* 系命令

命令		引数
get_variable	Xn, Ai	初出の一時変数 Xn
get_variable	Yn, Ai	初出の局所変数 Yn
get_value	Xn, Ai	既出の一時変数 Xn
get_value	Yn, Ai	既出の局所変数 Yn
get_constant	C, Ai	アトミックな定数 C
get_nil	Ai	nil ('[]')
get_list	Ai	リスト・セル
get_vector	N, Ai	N 要素のベクタ

ローカル・スタック上のフレームである Environment は，m 個の局所変数 $\mathbf{Y_1} \sim \mathbf{Y_m}$ の他，**E** と **CP** の退避領域として使用される。PSI-II では更に，**B** と **LVLC** も Environment に退避される。一方 Choice Point には，**B, E, CP, G, TR,** 候補節のアドレス (**AP**) 及び $\mathbf{A_1} \sim \mathbf{A_n}$ が退避される。また PSI-II では **LVLC** も退避される。

命令は，*get, put, unify, control, indexing* の 5 グループに大別される。以下各グループについて概説するが，付録 C に命令の一覧と各命令の機能，PSI-II/PSI-III での実行サイクル数を示しているので，適宜参照されたい。また付録 D には，2.1 に示した例 elder_brother(秀忠, B) に関して，対応する WAM のコード，実行過程でのスタックの状態などを示している。

(1) get 系命令

get 系の命令はヘッド・ユニフィケーションを行うものであり，原則として一つのヘッド引数に一つの命令が対応する。i 番目の引数 $\mathbf{A_i}$ のユニフィケーションは，ヘッド引数の種類に応じて表 3-2に示す命令により行われる。即ち，get_variable は初出の変数に対応し，$\mathbf{A_i}$ を単に変数 $\mathbf{X_i}/\mathbf{Y_i}$ に代入する。get_value は既出の変数に対応し，$\mathbf{A_i}$ と $\mathbf{X_i}/\mathbf{Yi}$ の「一般的な」ユニフィケーションを行う。

get_constant, get_nil はヘッドに書かれた定数に対応し，$\mathbf{A_i}$ と定数とのユニフィケーションを行う。なお，PSI-II ではアトム及び絶対値が小さい整数のために；

```
get_integer   C, Ai
get_atom      C, Ai
```

を用意し，これらのユニフィケーションの高速化を図っている。

表 3-3: *put* 系命令

命令		引数
put_variable	Xn, Ai	初出の一時変数 Xn
put_variable	Yn, Ai	初出の局所変数 Yn
put_value	Xn, Ai	既出の一時変数 Xn
put_value	Yn, Ai	既出の局所変数 Yn
put_unsafe_value	Yn, Ai	既出の「危険な」局所変数 Yn
put_constant	C, Ai	アトミックな定数 C
put_nil	Ai	nil ('[]')
put_list	Ai	リスト・セル
put_vector	N, Ai	N 要素のベクタ

構造体のユニフィケーションは，get_list, get_vector* が行うが，その要素に関しては引続く *unify* 系の命令が実行する。

なお，get_variable 以外の命令では，ユニフィケーションが失敗することがあり，その場合にはバックトラック処理が行われる。即ち，最新の Choice Point に退避された情報に基づき，実行環境の復元と変数代入の無効化 (undo) が行われる。また，候補節への分岐，即ち Choice Point 中の **AP** が指示するアドレスへの分岐が行われる。従って，get_variable を除く *get* 系の命令は，潜在的な条件分岐命令であると言うことができる。

(2) put 系命令

put 系の命令はゴール引数の生成を行うものであり，原則として一つのゴール引数に一つの命令が対応する。i 番目の引数 $\mathbf{A_i}$ の生成は，ゴール引数の種類に応じて表 3-3に示す命令により行われる。即ち，put_vaiable は初出の変数に対応し，変数セルの生成／初期化を行い，それを指示する Reference Pointer を $\mathbf{A_i}$ に設定する。また，put_value は既出の変数に対応し，単に変数の値を $\mathbf{A_i}$ に代入する。但し，put_unsafe_value は「危険な」局所変数に対応するものであり，TRO のために用意されている。即ち，最後のゴールに関しては，ヘッドまたは構造体中に出現しない変数 Y に関する put_value が put_unsafe_value に置き換えられ，Y が未定義であれば†Y をグローバル・スタックに移動する。

put_constant, put_nil は定数であるようなゴール引数に対応し，指定された定数を単に $\mathbf{A_i}$ へ代入する。なお，PSI-II では *get* 系と同様に；

*[Warren 83] の "get_structure F, Ai" に相当する。

†正確には最新の Envrionment 中に存在している未定義変数である場合。

表 3-4: *unify* 系命令

命令		引数
unify_variable	Xn	初出の一時変数 Xn
unify_variable	Yn	初出の局所変数 Yn
unify_value	Xn	既出の一時変数 Xn
unify_value	Yn	既出の局所変数 Yn
unify_void	N	N 個の無名変数
unify_constant	C	アトミックな定数 C

```
put_integer   C, Ai
put_atom      C, Ai
```

を用意している。

ゴール引数として出現する構造体に対応する命令は put_list, put_vector* であるが，その要素の生成は引続く *unify* 系の命令が行う。

(3) unify 系命令

unify 系の命令は，構造体に関するヘッド・ユニフィケーションとゴール引数生成に用いられる。構造体の一つの要素が一つの命令に対応し，その種類に応じて表 3-4が用意されている。また，PSI-II では *get*/*put* 系と同様に；

```
unify_integer   C, Ai
unify_atom      C, Ai
```

を用意している。

unify 系の命令の動作は，構造体に対応する命令が *get* または *put* 系のいずれであるか，また *get* 系の場合には引数 $\mathbf{A_i}$ が構造体であったか未定義変数であったかによって異なる。

get 系でかつ $\mathbf{A_i}$ が構造体であった場合，*get* 系命令は構造体の最初の要素のアドレスをレジスタ S にセットする。また *unify* 系命令は「読出モード」で動作し，そのオペランドと S が指示するデータとのユニフィケーションを行い，S を次の要素に進める。なお，unify_variable 以外の命令ではユニフィケーションが失敗することがあるが，その場合には *get* 系命令と同様にバックトラック処理が行われる。

*[Warren 83] の "put_structure F, Ai" に相当する。

一方 *get* 系で $\mathbf{A_i}$ が未定義変数であった場合，変数にはグローバル・スタック・トップを指す構造体ポインタが代入される。同様に *put* 系の場合には，構造体ポインタがゴール引数 $\mathbf{A_i}$ に代入される。一方，*unify* 系命令の動作は「書込モード」となり，グローバル・スタックにオペランドをプッシュする。従って *get*/*put* 系命令が生成した構造体ポインタは，*unify* 系命令が生成した要素列の先頭を指すこととなり，結果的に構造体が生成される。

副構造体を持つ構造体のユニフィケーションやゴール引数生成は，副構造体を一時変数に置くことによって，スタックなどを使うことなく実現される。例えば，ヘッド・ユニフィケーション；

```
p(f(A,[g(B)|C])):- ...
```

は，概念的には；

```
p(f(A,V1)):- V1=[V2|C], V2=g(B), ...
```

に置き換えられ，以下の命令列によって実行される（A, B, C は局所変数 $\mathbf{Y_1} \sim \mathbf{Y_3}$ とする）。

```
get_vector      3, A1
unify_atom      f       % f(
unify_variable  Y1      %   A,
unify_variable  X2      %     V1)
get_list        X2      % V1=[
unify_variable  X2      %     V2|
unify_variable  Y3      %       C]
get_vector      2, X2   % V2=
unify_atom      g       %   g(
unify_variable  Y2      %     B)
```

同様に，ゴール引数の生成；

```
..., p(f(A,[g(B)|C])), ...
```

は，概念的には；

```
..., V1=g(B), V2=[V1|C], p(f(A,V2)), ...
```

に置き換えられ，以下の命令列によって実行される（A, B, C は既出の局所変数 $\mathbf{Y_1} \sim \mathbf{Y_3}$ とする）。

```
put_vector      2, X1   % V1=
unify_atom      g       %    g(
unify_value     Y2      %      B)
put_list        X2      % V2=[
unify_value     X1      %     V1|
unify_value     Y3      %       C]
put_vector      3, A1
unify_atom      f       % f(
unify_value     Y1      %   A,
unify_value     X2      %     V2)
```

(4) control 系命令

control 系の命令はゴールの呼出し，ユニット・クローズの実行完了，及び Environment の生成と除去を行う。ゴール呼出しは，最終ゴール以外に関しては復帰アドレスの **CP** へのセットを伴う `call` が，また最終ゴールに関しては単に分岐を行う `execute` が用いられる。また，ユニット・クローズの実行完了は `proceed` が行い，**CP** が保持する復帰アドレスへ分岐する。

Environment の生成は，2.4 で述べたように二つ以上のゴールを持つクローズに対してのみ行われる。このようなクローズに対しては，ヘッド・ユニフィケーションを行う前に，`N` 個の局所変数を持つ Environment 生成命令 "`allocate N`" が発行される。また Environment の除去は，最終ゴールの呼出しの直前に `deallocate` により行われ，TRO が実現される。

以上をまとめると，ゴールの数が n であるようなクローズに対しては，以下のような命令列が生成される。

$n = 0$: *get* + `proceed`
$n = 1$: *get* + (*put* + `execute`)
$n \geq 2$: `allocate` + *get* + (*put* + `call`) + ... + (*put* + `call`)
+ (*put* + `deallocate` + `execute`)

(5) indexing 系命令

indexing 系命令は，一つの述語に属する複数のクローズを結合するために用いられる。まず，基本的なクローズの関係である出現順序に基く OR の関係は；


```
    try_me_else     C1
    "codes for 1st clause"
C1: retry_me_else   C2
    "codes for 2nd clause"
C2: retry_me_else   C3
    "codes for 3rd clause"
 ⋮
Cn: trust_me
    "codes for n-th clause"
```

によって表現される。即ち，`try_me_else` は最初のクローズの実行の前に Choice Point を生成し，`trust_me` は最後のクローズの実行前に Choice Point を除去する。また `retry_me_else` は中間のクローズの実行に先立って，Choice Point 中の **AP** を次のクローズのアドレスに書換える。

一方 Clause Indexing は，引数レジスタ A_i が保持するデータのタイプまたはその値による多方向分岐命令と，非連続のクローズを結合する命令群によって行われる。多方向分岐命令には；

```
switch_on_term          Ai, Lv, Lc, Ll, Ls
switch_on_constant      Ai, Table
switch_on_structure     Ai, Table
```

の三種がある。`switch_on_term` は A_i が保持するデータが，未定義変数，アトミック，リスト，ベクタ，のいずれであるかによって，`Lv, Lc, Ll, Ls` に分岐する*。一方 `switch_on_constant/switch_on_structure` は，A_i が保持するアトミック・データの値，または構造体の「名前」と要素数に基きハッシュ・テーブル `Table` を索引して得られるアドレスへ分岐する。

さて，Clause Indexing によって唯一のクローズが選択される場合には，これらの命令の分岐先はそのクローズに対するコード，即ち `try_me_else` などの次の命令となる。しかし，選択対象が複数ある場合には，それらを結合するために；

```
try     Lab
retry   Lab
trust   Lab
```

が用いられる。これらの命令は `try_me_else` など 'me' が付いたものと同様の動作をするが，Choice Point の **AP** に自身の次のアドレスを設定し，`Lab` へ分岐する点が異なっている。

*ヒープ・ベクタのようにアトミック，（スタック）ベクタ，リストのいずれでもない場合は `Lv` に分岐する。

これらを用いて，*Quick Sort* の一部分である partition；

```
partition([],_,L,L):-!.
partition([X|L1],Y,[X|L2],L3):- X < Y, !, partition(L1,Y,L2,L3).
partition([X|L1],Y,L2,[X|L3]):- partition(L1,Y,L2,L3).
```

は，以下のようにコンパイルされる。

```
partition:
        switch_on_term  A1, C1a, C1, L1, fail
L1:     try             C2
        trust           C3

C1a:    try_me_else     C2a
C1:     "codes for partition([],_,L,L):-!."

C2a:    retry_me_else   C3a
C2:     "codes for partition([X|L1],Y,[X|L2],L3):- X<Y, !, ..."

C3a:    trust_me
C3:     "codes for partition([X|L1],Y,L2,[X|L3]):- ..."
```

また，[Warren 83] に示された例；

```
call(X or Y) :- call(X).
call(X or Y) :- call(Y).
call(trace):- trace.
call(notrace):- notrace.
call(nl):- nl.
call(X):- builtin(X).
call(X):- ext(X).
call(call(X)):- call(X).
call(repeat).
call(repeat):- call(repeat).
call(true).
```

は，図3-12のようにコンパイルされる。なお，このコンパイル方法は [Warren 83] に示されたものとは異なり，一つの述語に対して複数の Choice Point を生成せず，変則的な retry (_me_else) の使用によって，複雑な Clause Indexing を実現している。また，jump は無条件分岐命令，fail はバックトラックを行う命令である。

なお PSI-II では無用なハッシングの回避や，第一引数が変数であるクローズの決定的選択など，Clause Indexing の拡張も行っているが，これらに関しては 4.2.1 で述べる。


```
call:   try_me_else     C6a
        switch_on_term  A1, C1a, L1, fail, L2
L1:     switch_on_constant    A1, <trace:C3, notrace:C4, nl:C5>
L2:     switch_on_structure   A1, <or/2:L3>
L3:     retry           C1
        retry           C2
        jump            C6a
C1a:    retry_me_else   C2a
C1      "codes for call(X or Y):- call(X)."
C2a:    retry_me_else   C3a
C2:     "codes for call(X or Y):- call(Y)."
C3a:    retry_me_else   C4a
C3:     "codes for call(trace):- trace."
C4a:    retry_me_else   C5a
C4:     "codes for call(notrace):- notrace."
C5a:    retry_me_else   C6a
C5:     "codes for call(nl):- nl."
C6a:    retry_me_else   C7a
        "codes for call(X):- builtin(X)."
C7a:    retry_me_else   L4
        "codes for call(X):- ext(X)."
L4:     switch_on_term  A1, C8a, L5, L7, L8
L5:     switch_on_constant    A1, <repeat:L6, true:C11>
L6:     retry           C9
        trust           C10
L7:     trust_me
        fail
L8:     trust           C8
C8a:    retry_me_else   C9a
C8:     "codes for call(call(X)):- call(X)."
C9a:    retry_me_else   C10a
C9:     "codes for call(repeat)."
C10a:   retry_me_else   C11a
C10:    "codes for call(repeat):- call(repeat).
C11a:   trust_me
C11:    "codes for call(true)."
```

図 3-12: call(X) のコンパイル・コード

3.2.2.2 PSI-I 上での実験

PSI-IIの機械命令としてWAMの命令セットを導入するに当って，その効率を予測し，また必要なハードウェアを検討するために，PSI-IにWAMを実験的に実装した[Nakashima 87a]。

実験システムは，命令フェッチとデコードに重点を置いて設計した。まず，プログラム・カウンタは**pLAR**に割付け，インクリメント機能を有効に活用した。命令レジスタは**pDR**に割付け，命令コードとしてタグを用いることによって，命令をエミュレートするマイクロプログラム・ルーチンへの分岐をタグ多方向分岐で実現した。また，**WF**の間接アクセス領域に割付けた引数レジスタへのアクセスも，**pDR**による**WF**の間接アクセス機能を活用した。更に，エミュレーション・ルーチンの中での命令先行フェッチにより，命令フェッチ・オーバヘッドを最小化する工夫も行った。

この他，従来は**WF**のS1-Busにのみ読出可能な領域に割付けられていた制御レジスタを，S2-Busへの読出が可能な領域に移動してアクセスにかかる手間を削減した。また，3.2.6で述べるメモリ割当のチェック機構を試験的に導入し，動的メモリ割付けに伴うオーバヘッドの削減を図った。なお，組込述語については35種類のものを4.1で述べる手法に基き実装し，KL0特有の実行順序制御などに関しても4.3で述べる手法により必要な配慮を行った。

さて，実験システムの性能をいくつかのベンチマーク・プログラムによって測定したところ，5.1.1で述べるようにPSI-Iの1.7 ~ 3.3倍であることが明らかになった。この性能差には，前述のような様々な実現手法の工夫によるものが含まれているが，基本的にはマイクロ・インタプリタ方式とコンパイル方式の違いが現れたものと考えられる。そこで，appendのRecursive Clause；

```
append([X|L1],L2,[X|L3]):- append(L1,L2,L3).
```

の実行過程がどのように異なっているかを比較してみた。但し，第一，第二引数はそれぞれリストであり，第三引数は未定義変数であるとする。

まずWAMの命令コードとKL0の内部表現を「命令的」に記述したものを，図3-13と図3-14にそれぞれ示す。WAMのコードでは，巧妙な一時変数の割付けによるFirst Goal Optimizationが極めて効果的に行われていることに注目すべきである。即ち，一時変数L1, L2, L3をX_1, X_2, X_3と同じレジスタであるA_1, A_2, A_3に割付けたことにより，これらに対するput_value命令が全て省略されている。また，L2に関してはget_variableも省略することができるため，ヘッド・ユニフィケーションと引数生成の全てが省略されている。


```
append: switch_on_term  A1, C1a, C1, C2, fail

C1a:    try_me_else     C2a     % append(
C1      get_nil         A1      %        [],
        get_value       A2, A3  %           L,L
        proceed                 %              ).

C2a:    trust_me                % append(
C2      get_list        A1      %        [
        unify_variable  X4      %         X|
        unify_variable  A1      %           L1],
                                %              L2,
        get_list        A3      %                 [
        unify_value     X4      %                  X|
        unify_varable   A3      %                    L3]):-
        execute         append  %     append(L1,L2,L3).
```

図 3-13: *append*のWAMコード

```
append: pred_header     Args=3
        clause_header   Args=3,Lvars=0,Gvars=0,  % append(
                        Unit_Clause,Have_Alternatives
        alternative     C2
        atom            []                      %     [],
        lvar            #1                      %     L,
        lval            #1                      %     L).

C2:     clause_header   Args=3,Lvars=0,Gvars=3,  % append(
                        Normal_Clause,No_Alternatives
        alternative     none
        list            L1                      %     [X|L1],
        lvar            #1                      %     L2,
        list            L2                      %     [X|L3]):-
        last_goal       append                  %   append(
        gval            #2                      %     L1,
        lval            #1                      %     L2,
        gval            #3                      %     L3).

L1:     gvar            #1                      %   [X|
        gvar            #2                      %     L1]
L2:     gval            #1                      %   [X|
        gvar            #3                      %     L3]
```

図 3-14: *append*のKL0内部表現

さてappendが呼出されると，WAMではswitch_on_termによりA_1がリストであることが判定され，Choice Pointを生成することなくラベルC2への分岐が行われる。一方，KL0のマイクロ・インタプリタ（以下μIntと呼ぶ）では，引数のタイプによるClause Indexing機能がないため，第一クローズが実行される。但し，シャロー・バックトラックの最適化[Yokota 84]によって，Choice Pointは生成されない。しかし，候補節があること(Have_Alternatives)やそのアドレスは，内部的に記憶される。その後，第一クローズのヘッド・ユニフィケーションが行われて，その失敗によりバックトラックが発生するが，その過程については省略する。

次にWAMでは直ちに第一引数のユニフィケーションが開始されるが，μIntではクローズ実行の前処理が行われる。即ち；

(1) 引数の数が3であることを知り，ローカル・スタックを3ワード分だけ伸ばせることをチェックする。また，引数の個数を「引数カウンタ」に記憶する。
(2) 引数以外に局所変数がないことを知る。
(3) コントロール・スタックを1フレーム分だけ伸ばせることをチェックする。
(4) 大域変数の数が3であることを知り，グローバル・スタックを3ワード分だけ伸ばせることをチェックし，3個の未定義変数セルをプッシュする。
(5) ユニット・クローズではないことを記憶する。
(6) 他に候補節がないことを記憶し，次のワードをスキップする。

これらの内，(4)での領域チェックはWAMにおいても必要な処理であり，実際の処理系では必ず実施される。またこの時点でグローバル・スタックのプッシュを行うのは，構造体複写法を採用しているためである。一方，(1)～(3)については，ゴールが一つしかない場合には結局不要となる処理であるが，そのような情報は与えられていないため（後にその事実を発見する）無駄な処理を行ってしまう。また，(5)と(6)については，この情報が必要な時期はヘッド・ユニフィケーション後であるにも関わらず，提示されている位置が悪いため，「記憶する」という処理が必要となる。更にこれら情報が1ワードの中に詰め込まれているために抽出処理が必要なことや，(2)や(6)のように「何もしなくて良い」ということが陽に提示されていることに注意しなければならない。

さて，第一引数のユニフィケーションであるが，構造体複写法と共有法の違いが原因となって，WAMではレジスタを用いることができる処理のために，μIntではグローバル・スタックへのアクセスが必要となる。また，リスト・セルの要素がμIntでは離れた場所，即ちスケルトンL1に記述されていることも，オーバヘッドの要因となっている。

次に，第二引数のユニフィケーションについては，WAMでは命令自体が省略されているためいかなる処理も行われない。一方μIntでは「初出の局所変数」という情報が提示されているが，これは「何もしなくて良い」ということを意味している。従って，このワードはスキップされるが，「何もしなくて良い」ということを知る必要がある。

第三引数のユニフィケーションについては，WAMでは個々の要素に関する処理をしながらグローバル・スタック上に構造体が生成される。μIntでは要素に関する処理は一切不要であり，スケルトンL2の内部は参照されない。但し，2.2で述べた2ワードのMoleculeが生成されるため，グローバル・スタックへの書込回数は結果的に等しくなる。

ヘッド・ユニフィケーションが完了すると，WAMではFirst Goal Optimizationの効果で，直ちにexecuteが実行されて，appendの処理は完了する。一方μIntでは，まず引数を全てユニファイしたことを，「引数カウンタ」が0になったことによって知る。このため，引数カウンタのデクリメントとゼロ判定が，引数を一つ処理するたびに必要となる。次に，ユニット・クローズではないことを「思い出し」，次のワードを読出す。その結果，呼出すゴールが最後のものであることを知り，それまでにゴールを呼出していないことと候補節が存在しないことを「思い出し」，ローカル・スタック／コントロール・スタックにフレームを生成する必要がないと判断する。

これらの処理が終わると引数の準備を始めるが，まず何個引数を準備すれば良いかを呼出す述語のヘッダを読んで知り，それを引数カウンタにセットする。そして，フレーム・バッファの切換を行った後，WAMの*put*系命令と同様に引数を順次フレーム・バッファに準備する。その際，引数カウンタをデクリメントして終了チェックを行い，0になった時点で呼出す述語の第一クローズの処理を開始する。

以上のように，KL0のマイクロ・インタプリタの処理はかなり複雑であるが，この原因は必要な情報が必要な時点で提示されていないことにある。この結果，クローズ・ヘッダに記された情報を保存し，またそれを必要になった時点で「思い出す」という，無駄な処理が発生している。同様の無駄は，ゴール引数の準備と実行すべき述語の指示が，WAMとは逆になっていることからも発生している。更にこれに関連して，「何もしなくて良い」という不必要な情報が提示されることも，高速処理の妨げになっている。

また，「命令的」な処理がデータとして提示されるために，その解釈が処理の文脈に依存してしまうのも欠点である。例えばコードを指示するポインタは，ゴールを表現するだけではなく，「コード型のデータ」がヘッド／ゴール引数として出現した場合にも使用されるため*，ユニフィケーションや引数準備の終了判定に使用することができない。従って，引数カウンタのゼロ・チェックという二重の判定が必要となっている。

この他，構造体複写法が共有法に比べて単純であることも含め，WAMの方が高性能の処理系を構築できることが明らかであるという結論に達した。

*実際にはまずありえない。

3.2.2.3 命令の内部表現

命令をどのようにビット・パターンで表現するかは，ハードウェアの構成に大きく影響する。

まず第一の選択肢は，命令をワード単位とするか，より小さな単位（例えばバイト単位）とするかであるが，ハードウェア構成の単純化を優先してワード単位とすることとした。この選択の背景には，論理型言語の処理ではデータをバイト単位でアクセスすることが稀であるため，バイト・アクセス機構を用意しても命令フェッチ専用になってしまうという事実がある。また，ワード単位の方式の欠点であるコード領域の大きさについては，命令の機能が高いことや，4.4で述べる複合命令の使用により，さほど重大な問題にはならないと判断した。実際 append のコード量は，命令コードを8ビット，レジスタ指定を4ビット，分岐アドレス指定を16ビットとした時，ワード単位の場合は15w，バイト単位の場合は（タグは使用できないとして）10wとなり，極端な差異はない。更に，複合命令を導入するとワード単位でも11wとなり，その差が一層縮まる。なお，PSI-I における KL0 の内部表現は（スケルトンを除いて）14wであり，これも大きな差異はない。

次に命令コードのビット数であるが，4.1で述べるように組込述語を命令として実現する最適化のために，150種類を超える組込述語をコード化する必要があり，充分に大きな値とすることが望まれた。また，3.4で述べるように，PSI-II 上で KL0 と KL1 の処理系を同時に動作させる pseudo Multi-PSI/v2 の実現や，LISP など他言語の処理系構築も予想されたため，命令コードに余裕を持たせる必要があった。そこで，8ビットの命令コードを持つ，4種類の「命令クラス」を設けることによって，最大1024種類の命令を実現できるようにした。この内 KL0 の処理には，基本命令と組込述語のための2つの命令クラスが使用される。

オペランドの表現は，命令長に余裕があることから8ビット単位とした。即ち；

8-bit：引数レジスタ番号，局所変数番号，整数即値
16-bit：自己相対アドレス，整数即値
24-bit：アトム即値
40-bit：絶対アドレス，一般データ

とした。なお，実用的なプログラムでは多引数となることがしばしばあるため，引数レジスタは32個設けることとした。また，アトム即値とレジスタ番号の双方を使用する命令では，例外的に命令コードを3ビットとし，32ビットで命令コードとオペランドを表現できるようにした。

以上をまとめると，命令の表現形式は図3-15に示すものとなる。

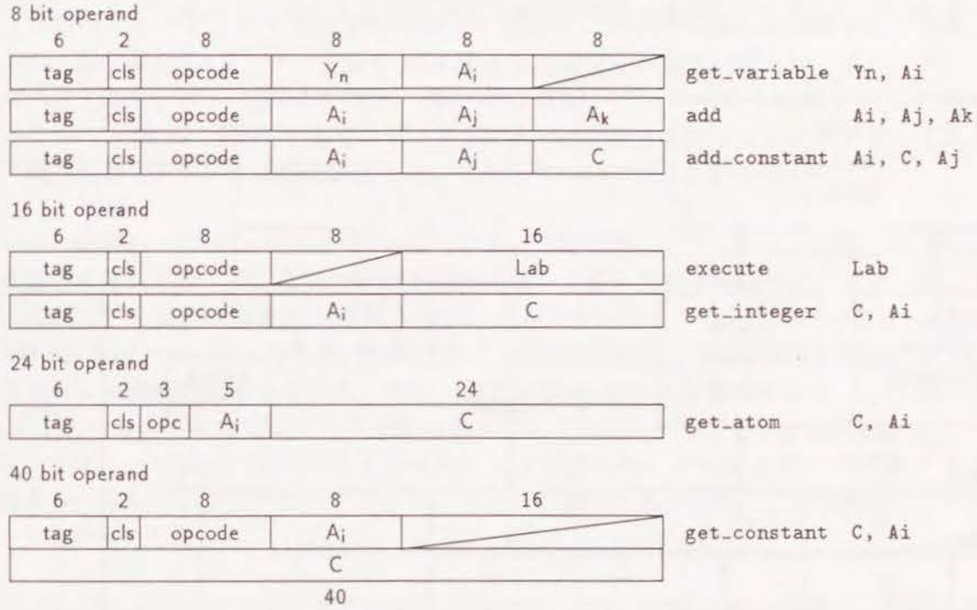


図 3-15: 命令の表現形式

3.2.3 ハードウェア構成

PSI-IIのハードウェアは，図3-16に示す構成となっており，マシン・サイクルは155 nsである*。

主記憶（物理記憶）はPSI-Iと同様に，7ビットのECCチェック・ビットが付加された，40ビット×32 Mw（最大）の構成となっている。但し拡張性を考慮して，物理アドレスを26ビットとして，64 Mwまでの容量に対応できるようにしている。

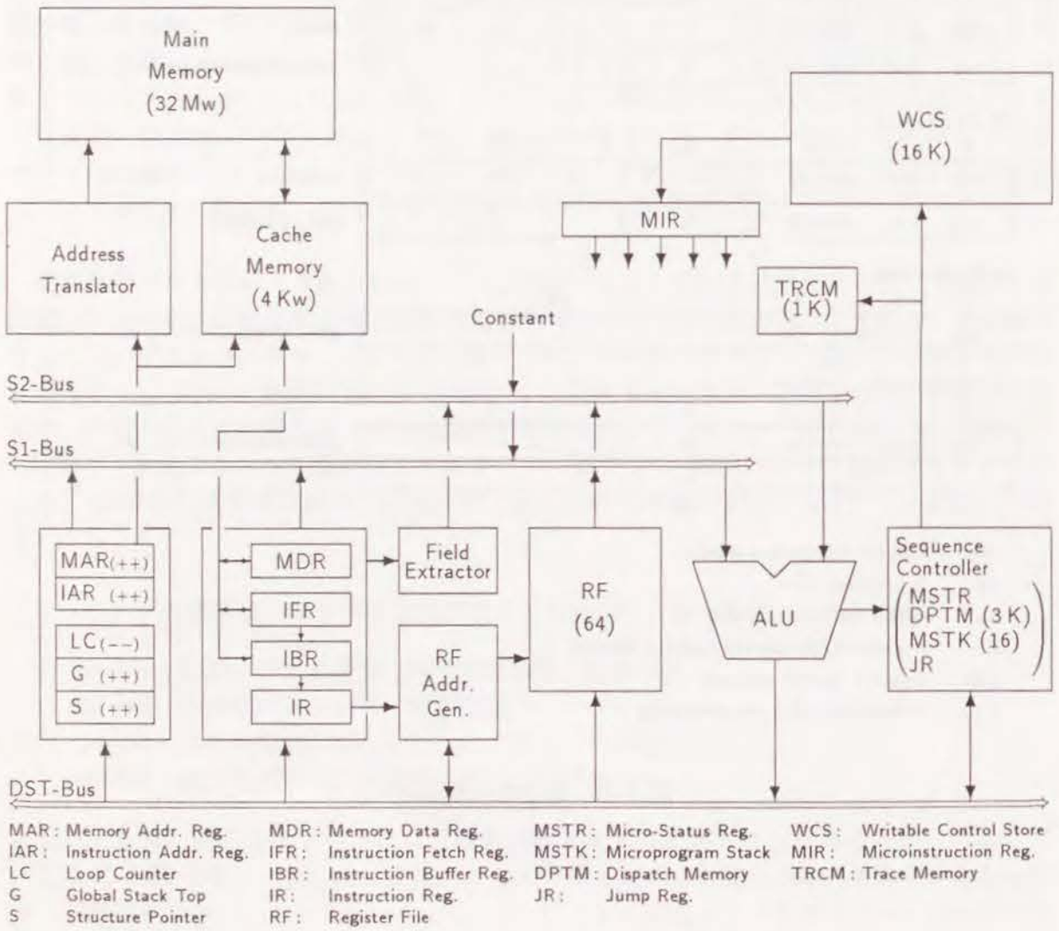


図 3-16: PSI-II のハードウェア構成

*最初のバージョンでは200 nsであった。

また，キャッシュ・メモリとアドレス変換機構の存在もPSI-Iと同様であるが，その構成は若干変化している。まず，キャッシュ・メモリはハードウェア量の削減のために，4 Kwのダイレクト・マッピング方式としたが，これによる性能の低下は5.3.2に示すように軽微であることが評価によって明らかになっている。また，アドレス変換機構については，プロセスごとに32ビットの論理アドレスを与える多重論理空間の採用や，メモリ割当のチェック機構の導入が，主な変更点である。なお，これらについては3.2.6で詳しく述べる。

メモリ系とCPUとのインタフェースは，命令フェッチのために大きく変更された。まず，二個のインクリメント機能を持ったアドレス・レジスタを備えた点はPSI-Iと同じであるが，命令用を**IAR**，データ用を**MAR**として機能を明確に分離した。また，データ・レジスタを命令／データ共用の**IFR**，命令用の**IBR**，データ用の**MDR**の三つに増加するとともに，**IFR**，**IBR**と命令レジスタ**IR**をパイプライン・レジスタとすることによって，強力な命令フェッチ機構を実現した。これらについては3.2.4で述べる。

ALUの入力／出力バスであるS1-Bus, S2-Bus, DST-Busと，それらに接続されるレジスタの構成に関しては，PSI-Iでの評価や経験に基いて様々な変更を加えた。まず，PSI-IのWFに相当する**RF**の容量を64 wに削減し，高速のマルチポート・メモリであるAm29334 [AMD 85]を用いて実現した。この結果**RF**の全領域に関して，S1-Bus/S2-Busの双方に対する2ポート読出が可能となった。また，**RF**の前半32 wに引数レジスタ$\mathbf{A_i}$を割付け，3.2.4で述べるように命令レジスタ**IR**のオペランド・フィールドを用いた間接アクセス機能を，2ポートの読出と書込のいずれに対しても用意した。また後半32 wは直接アクセス領域とし，制御用レジスタやスクラッチ・パッドに割当てた。なお，定数領域やトレイル・バッファは廃止した。

S1-Bus/S2-Busの双方には**RF**の他に，**MDR**, **IFR**, **IBR**, **IR**を接続し，頻繁に行われるメモリ・データや命令語に関する操作に配慮した構成とした。また，バイト単位のシフト／マスク機能をこれらのレジスタとS2-Busの間に配置することにより，ALU周辺に関わる遅延時間を大きくすることなく，オペランド抽出などの機能を提供することができた。

一方S1-Busには，アドレス・レジスタである**MAR/IAR**の他に，3つのレジスタ**LC**, **G**, **S**を接続した。**LC**はループ制御用のダウン・カウンタであり，**G/S**は*unify*系の命令の高速処理のためにアップ・カウンタとした。また，S1-Bus/S2-Busの双方に対して定数出力を可能とし，特にS2-Busに関しては強力な定数生成機構を用意して，WF定数領域の廃止のダメージを最小化した。なお，上記以外のレジスタ／テーブル類については，全てDST- Busにのみ接続し，S1-Bus/S2-Busの物理的線長や負荷を最小化した。

ALUは高機能かつ高速な演算チップAm29332 [AMD 85]を用いて構成した。この結果，レジスタの値を桁数とするシフト演算が可能となった他，乗除算に関する操作も強化された。なお，バイト・スワップ機構は廃止した。

マイクロプログラムの順序制御については，命令デコードのための多方向分岐機能を強化した他は，PSI-I とのハードウェア構成上の差異は少ない。しかし，3.2.7 で述べるように，マイクロ命令の構成の面で分岐機能を使いやすくするための配慮が成されている。また，マイクロプログラム・サブルーチンのためのスタックを 1 Kw から 16 w に圧縮し，マイクロプログラム・アドレスのトレース用メモリである **TRCM** を多方向分岐テーブル **DPTM** と同じメモリ素子上に同居させるなど，小型化のためにメモリ素子の実装量を削減する工夫を行った。

ハードウェアの実装は，以下に示す 8 種類の 2 μm CMOS ゲートアレイを中心に行った。

MCU キャッシュ・メモリ制御
MAU メモリ・アドレス
MDU メモリ・データ
RG1 レジスタの値部（16 ビット・スライス，2 チップ使用）
RG2 レジスタのタグ部
SEQ 順序制御
BCU I/O バス・インタフェース
CCU システム制御

これらの LSI の平均ゲート搭載量は約 5,700 ゲートである。また，Am29332 を含めた総ゲート数は 58,000 ゲートであり，PSI-I の論理素子の約 1,700 チップ分に相当する。

LSI の構成については，1 チップのゲート数／入出力ピンの制限やバスの物理的線長の短縮のために，レジスタの分割実装や重複実装を行った。例えば，**IFR** は；

- S1-Bus/S2-Bus への出力
- メモリへの書込データ供給
- 命令パイプライン・レジスタ
- タグ多方向分岐

の 4 つの機能がある。これら全てを一つの LSI チップ上に実装すると，多量の論理回路と入出力ピンを必要とする。そこで，以下のような実装を行った。

RG1 ビット 31 ～ 16 / 15 ～ 0 ⇒ S1-Bus/S2-Bus への出力
RG2 ビット 39 ～ 0 ⇒ 命令パイプライン・レジスタ，タグ多方向分岐
MDU ビット 39 ～ 0 ⇒ メモリ書込みデータ

このような分割／重複実装は他のレジスタについても行っており，実装量と遅延時間の両面で大きな効果があった。

この他，メモリ素子については，個々のテーブル／バッファの容量削減，実装上の工夫，大容量素子の採用などにより素子数の削減を図り，PSI-I では 190 チップ使用した SRAM

を 70 チップに減らすことができた。これらの結果，CPU の実装規模はプリント基板 3 枚となり，PSI-I の 1/4 に圧縮された。更に，主記憶を両面実装可能な 1 Mbit の DRAM を用いて構成し，入出力制御装置や機器に関しても小型化を図った結果，筐体の大きさは高さ 600 mm，幅 400 mm，奥行き 612 mm となり，PSI-I の約 1/6 の容積となった。

3.2.4 命令フェッチ / デコード機構

PSI-IIではWAMに基く機械命令を導入したため，PSI-Iに比べて命令を取扱うための機構の重要性が著しく増加した。まず，命令フェッチについては，3.2.2で述べた試験実装の結果から，先行フェッチの重要性が明らかになった。デコードについては，命令コードによる多方向分岐が基本となるが，*unify*系の命令における実行モードや，割込処理に関する配慮も必要であることが判明した。また，引数レジスタのアクセスを中心としたオペランド操作のための機構も，性能に大きく影響するものである。

以下，各々の機構について，その特徴や性能に与える効果を詳しく論ずる。

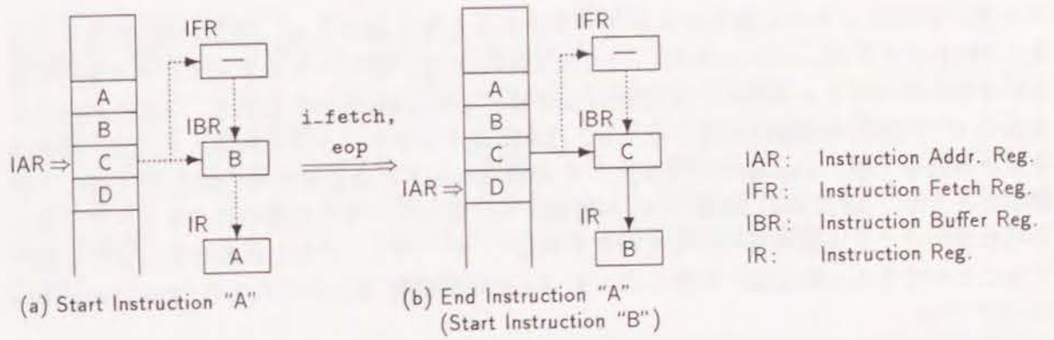
3.2.4.1 命令フェッチ

機械命令のエミュレーションにおいて，命令フェッチのオーバヘッドを最小化することは極めて重要である。PSI-IIの命令においても，レジスタ転送のみを行う`get_variable`など，操作自体が非常に単純なものが少なくなく，操作と命令フェッチのオーバラップ効果は極めて大きいものと考えられる。

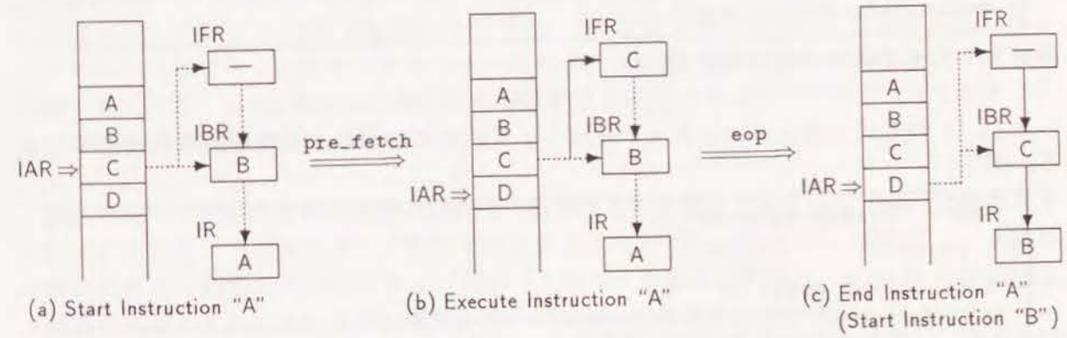
さて，命令フェッチ・オーバヘッドを除去するための単純な発想としては，自律的に動作する命令フェッチ機構の導入がある。しかしこのような機構は，他の機構との同期，メモリ・アクセスの競合の解決など，制御が複雑になるという欠点がある。そこでPSI-IIでは，簡単な命令パイプライン・レジスタをマイクロ命令で制御する方式を採用し，機構の単純化と命令フェッチ・オーバヘッドの最小化の双方を実現した。命令パイプライン・レジスタは，図3-17に示すように，**IFR**, **IBR**, **IR**の三つのレジスタから構成されている。これらの内**IFR**/**IBR**には，キャッシュ・メモリのデータバスが接続されており，プログラム・カウンタ**IAR**をアドレスとしてフェッチした命令を受取ることができる。また，**IBR**は命令デコードのために多方向分岐用テーブル**DPTM**に命令コードを供給し，**IR**は引数レジスタ・アクセスのための**RF**のアドレス生成などに用いられる。

さて，基本的な命令フェッチは図3-17(1)に示すように行われる。まず，命令Aの実行開始時点では，**IR**にAが，**IBR**に次の命令Bがそれぞれセットされており，**IAR**は更に次の命令Cのアドレスを保持している。命令Aのためのマイクロプログラム・ルーチンの最後のマイクロ命令では，命令終了操作`eop`が発行され，**IBR**が保持する命令Bの命令コードによる多方向分岐と，Bの**IR**への移動が行われる。同時に命令フェッチ操作`i_fetch`が行われて，**IAR**が指示する命令Cが**IBR**に格納されるとともに，**IAR**はインクリメントされる。

この操作はメモリ・アクセスを除く，全ての操作と並行して実行することができる。特



(1)



(2)

図 3-17: PSI-IIの命令フェッチ機構

に，条件付きの eop+i_fetch が可能であり，例えば；

```
if (X.tag == int) { i_fetch() , eop() ; }
raise_exception() ;
```

のように，データ・タイプのチェックなど様々な局面で使用される。また，この方法によってフェッチを行う命令の処理では，IFR をデータ・レジスタとして使用することができる。

一方，最後のマイクロ命令がメモリ・アクセスを伴う場合には，図 3-17(2) に示すように，IFR がプリフェッチ・バッファとして用いられる。即ち，メモリ・アクセスを行わないいずれかのマイクロ命令で，先行フェッチ操作 pre_fetch が行われて，IAR が指示する命令 C が IFR に格納されるとともに，IAR がインクリメントされる。その後，最後のマイクロ命令で単に eop 操作を行うと，B の命令コードによる多方向分岐と B の IR への移動とともに，IFR から IBR へ C が移動する。従って，命令処理中にメモリ・アクセスを行わないマイクロ命令が一つでも存在すれば*，オーバヘッドなしに命令フェッチを実行することができる。例えば，引数レジスタ A_i を局所変数 Y_n に代入する "get_variable Yn,Ai" では；

```
MAR = E + n , pre_fetch() ;
MDR = A[i] ;
write(MAR, MDR) , eop() ;
```

のように pre_fetch が用いられる。

さて，この方式で命令フェッチ・オーバヘッドが生ずるのは，分岐を行う命令のみである。即ち；

```
IAR = "Branch Address" ;
i_fetch() ;
i_fetch() , eop() ;
```

のように，分岐先とその次の二つの命令をフェッチしなければならないため，1 ステップのオーバヘッドが生じる。このような命令の出現頻度は約 25% と比較的多いが，その内 execute 以外のものでは二回の命令フェッチを何らかの操作と並行に行うことができる。従って，execute の実行頻度 2% とそのステップ数 3 から，命令フェッチのオーバヘッドは 1% 以下であると結論できる [Nakajima 87, Tateno 89]。

*KL0 の処理系では必ず存在する。

3.2.4.2 命令デコード

命令のデコードは，基本的には eop 操作により，IBR の命令コード・フィールドをアドレスとして DPTM をアクセスし，命令処理を行うマイクロプログラム・ルーチンへの分岐アドレスを得ることによって行われる。但しその際に，割込チェックと *unify* 系命令のためのモード判定が，並行して行われる。

まず eop 操作の時点で，割込フラグ・レジスタ ISTR とマスク・レジスタ IMSKR の論理積が 0 でないと，DPTM を索引して得たアドレスではなく，割込処理マイクロプログラム・ルーチンへ分岐する。割込処理ルーチンでは割込フラグを解析して適切な処理を行うが，必ずしもプロセス切換を伴うソフトウェア・レベルでの割込とする必要はない。即ち，何らかの処理を行った後，割込チェックを抑止した eop 操作によって，命令を継続実行することができる。従って，割込を隠された条件分岐として使うことができ，3.1.6 で述べるメモリ割当チェックや，4.3.1 で述べる様々な例外処理の効率的な実現のために用いられている。

モード判定操作は，ステータス・レジスタ MSTR のフラグ IMODE と，DPTM を索引して得られたアドレスのビット 1 の論理積をとることにより行われる。まず，get_list や get_vector は，引数がリストやベクタである時には IMODE をオフにし，未定義変数である時にはオンにする*。その後 eop 操作を行うと，*unify* 系の命令に対応する DPTM のエントリについてはビット 1 がオンになっているため，IMODE の値によって異なるアドレスへ分岐する。一方，*unify* 系以外の命令では DPTM の対応するエントリのビット 1 をオフとし，IMODE に無関係に分岐が行われる。

割込やモード判定のための機構の効果は，以下のように推定される。まず，一命令あたりのマイクロプログラム・ステップ数は平均 5 であるので [Yoshida-H 87]，割込チェック機構がない時には一命令あたりのステップ数が 0.5 増加すると仮定すると，約 10% の性能向上と考えられる。また，モード判定機構がない時には *unify* 系命令の処理がやはり 0.5 ステップ増加するとすれば，*unify* 系の実行頻度が 34.6% を占める [Tateno 89] に示された自然言語処理プログラムでは，約 3.5% の性能向上となる。これらの性能向上率は，極めて簡単なハードウェアで実現されたことを勘案すると，満足のできる値であると考えられる。

3.2.4.3 オペランド操作

オペランド操作の中で最も重要なものは，引数レジスタ，即ち RF のアクセスである。3.1.2 で述べたように引数レジスタ番号の指定は，命令語の各バイトに記述される。そこ

*それ以外ならばバックトラックする。

で，命令レジスタ IR の各バイト（上位から r0 ～ r3 フィールドと呼ばれる）を，RF の S1-Bus/S2-Bus への読出，及び DST-Bus からの書込アドレスとして使用できる構成とした。従って，引数レジスタ間の転送を行う "get_variable Xn,Ai" は；

```
RF[IR.r1] = RF[IR.r2] , i_fetch() , eop() ;
```

のように，1サイクルで実行することができる。なお，この機能の使用頻度は 27% と極めて高く，ハードウェアが有効に活用されていることが明らかになっている [Yoshida-H 87]。

この他，RF の S1-Bus への読出と DST-Bus への書込では，インクリメント機能を持ったレジスタ RFAR をアドレスとして使用することができ，Choice Point の生成やバックトラック時の引数レジスタの復元に用いられる。また，RF に関連するレジスタとして，A_i の書込を行うとビット i がオンになるレジスタ RVFR がある。RVFR をバックトラック時などにクリアしておくと，各引数レジスタが意味のあるデータを持っているか否かを判別することができ，ガベージ・コレクションのルート判定などに用いられる。

もう一つのオペランド操作機能は，IR のフィールド抽出である。この操作は，S2-Bus に接続されたバイト単位のシフト／マスク機構を用いて行なわれ，IR の下位 8, 16, 24 ビットの符号／ゼロ拡張による抽出と，任意のバイトの抽出が可能である。これらの機能を用いて，自己相対分岐アドレスの計算は；

```
IAR = IAR + sign_extend(IR & 0xffff) ;
```

と実現される。また，局所変数 Y_i のアドレスは；

```
MAR = E + zero_extend((IR >> 16) & 0xff) ;
```

のように計算することができる。なお，シフト／マスク機能は MDR, IFR, IBR にも適用することができ，一般のデータのフィールド抽出などに用いられる。

3.2.5 タグ・アーキテクチャ

PSI-II タグ・アーキテクチャは，演算系におけるタグ／値の分離やタグ即値の生成，順序制御系におけるタグによる二方向／多方向分岐など，PSI-I をほぼ踏襲したものとなっている。しかし，PSI-I での評価や経験に基き，特にマイクロ命令の設計に際して，他のハードウェア機構との並列動作を充分に配慮した。

3.2.5.1 演算系

PSI-I と同様，ALU の入出力パスである S1-Bus/S2-Bus/DST-Bus は，全てタグを持っている。また，整合性や統一性を意識して，S1-Bus/S2-Bus に接続されているレジスタには全てタグを持たせることとした。

タグに関する演算／転送の基本機能である；

- S1-Bus のタグ部または即値の DST-Bus のタグ部への転送
- S1-Bus と S2-Bus のタグ部の比較

に関しても，PSI-I とほぼ同様である。しかし，PSI-I ではマイクロ命令の構成上の問題で不可能であった，タグ即値生成と分岐の同時実行を可能とした結果，タグ即値の使用頻度は PSI-I の 2 ～ 3% から 15% へと，飛躍的に向上した。例えば，整数を引数レジスタに代入する "put_integer C,Ai" は；

```
A[i].tag = int , A[i].value = sign_extend(IR && 0xffff) ,
        i_fetch() , eop() ;
```

のように1ステップで実現することができる。

またタグの一致比較についても，PSI-I にはなかった値部の比較結果を含めた分岐条件，即ち；

```
if (MSTR.zero && MSTR.same_tag) ...
```

を設けることにより，有効な活用を促した。

なお，タグ転送／演算に関する補助的機能については，タグ部の ALU への入力は廃止し，MDR のタグ部をソース／デスティネーションとするものに転送のみに限定して，ハードウェアの削減を図った。但し，ガベージ・コレクタによる GC ビットの操作に配慮して，MDR の GC ビットをソース／デスティネーションとする転送も用意した。

3.2.5.2 順序制御系

順序制御系については，即値との比較対象をS1-Busのタグとし，RFのタグも比較対象とできるようにしたのが最大の改良点である。この結果，例えば“get_list Ai”は；

```
S = A[i] , if (A[i].tag == list) { i_fetch() , eop() ; }
```

のように，A_iがリストであれば1サイクルで実行することができる。

また，タグと即値の比較結果が「等しい」時の分岐の他に，「等しくない」時の分岐を設けて，組込述語のタイプ判定などの便宜を図った。更に，タグの上位4～7ビットのみの比較も設け，データ型集合の判別操作を可能とした。この機能は特にKL1処理系における，MRB*を除いたタグ判定に有効であった。

これらのように，タグと即値の比較機能を充実した結果，PSI-Iでは5%程度であった使用頻度が12%に増加し，改良の効果が明かとなった[Yoshida-H 87]。

DPTMを用いたタグ多方向分岐に関しては，PSI-Iのそれとほぼ同様であるが，バンク数を12から32に増やすとともに，ベースアドレスを自己相対だけではなく絶対アドレスによっても指定できるようにし，マイクロプログラムの割付作業が容易に行えるようにした。

*付録B参照。

3.2.6 メモリ・アーキテクチャ

3.2.6.1 アドレス変換機構

アドレス変換機構に関する重要な変更点は，プロセスごとに32ビットの論理アドレスを与える多重論理空間の採用と，メモリ割当チェックのための「グレイ・ページ」方式の導入である。

PSI-Iでは3.1.6で述べたように，エリア番号の中にプロセスのIDが含まれており，各プロセスには固有の論理アドレスが与えられる。従って，プロセスの数はエリア番号のビット数によって，最大63に制限される。また，一つのエリアに与えられる論理空間は最大16 Mwであるが，この値が将来に渡って充分なものであるとは必ずしも断言できない。

そこでPSI-IIでは，各プロセスに32ビットの論理空間を与え，プロセスごとに異なるアドレス変換を行うことによって，プロセス数の制限を無くすこととした。またこの結果，一つのエリアの大きさは最大512 Mwに拡大され，長期的な使用に耐え得るアーキテクチャとなった。

具体的なアドレス変換機構は，PSI-Iと同様の構成となっている。即ち，3ビットに圧縮されたエリア番号により，8エントリのページマップ・ベースが索引され，その結果と論理ページ番号を加算した値がページマップのエントリ・アドレスとなる。なお，8つのエリアの内0～3はプロセス共有領域，即ちヒープ及びシステム領域であり，4～7がプロセス固有のスタックに割付けられている*。従ってプロセス切換の際には，ページマップ・ベースの内のエントリ4～7のみを書換えれば良い。

ページマップの容量は，物理記憶空間を16 Mwから64 Mwに拡大したのに伴い，物理ページの最大量の1.5倍である96 Kページ分とした。なお，PSI-Iではページマップ上でエリアが衝突するのを避けるために，ページマップを物理記憶空間の2倍の容量としていたが，1.5倍でも衝突頻度は無視できる値であることが判明したため，ハードウェア量の圧縮のために削減した。

なお，多重論理空間導入の影響は，ガベージ・コレクションにも現れている。PSI-Iのガベージ・コレクションは[Morris 79]に示された逆転ポインタを用いる方式を基本としているが，多重論理空間の採用によってスタックからヒープへのポインタを単純に逆転することができなくなった。そこで，ガベージ・コレクション中に「リング」という領域をスタックから指されているヒープ上のデータの数だけ確保し，ポインタの逆転をリングとヒープの間で行う方法を開発した[Tateno 87]。

*1エリアは未使用。

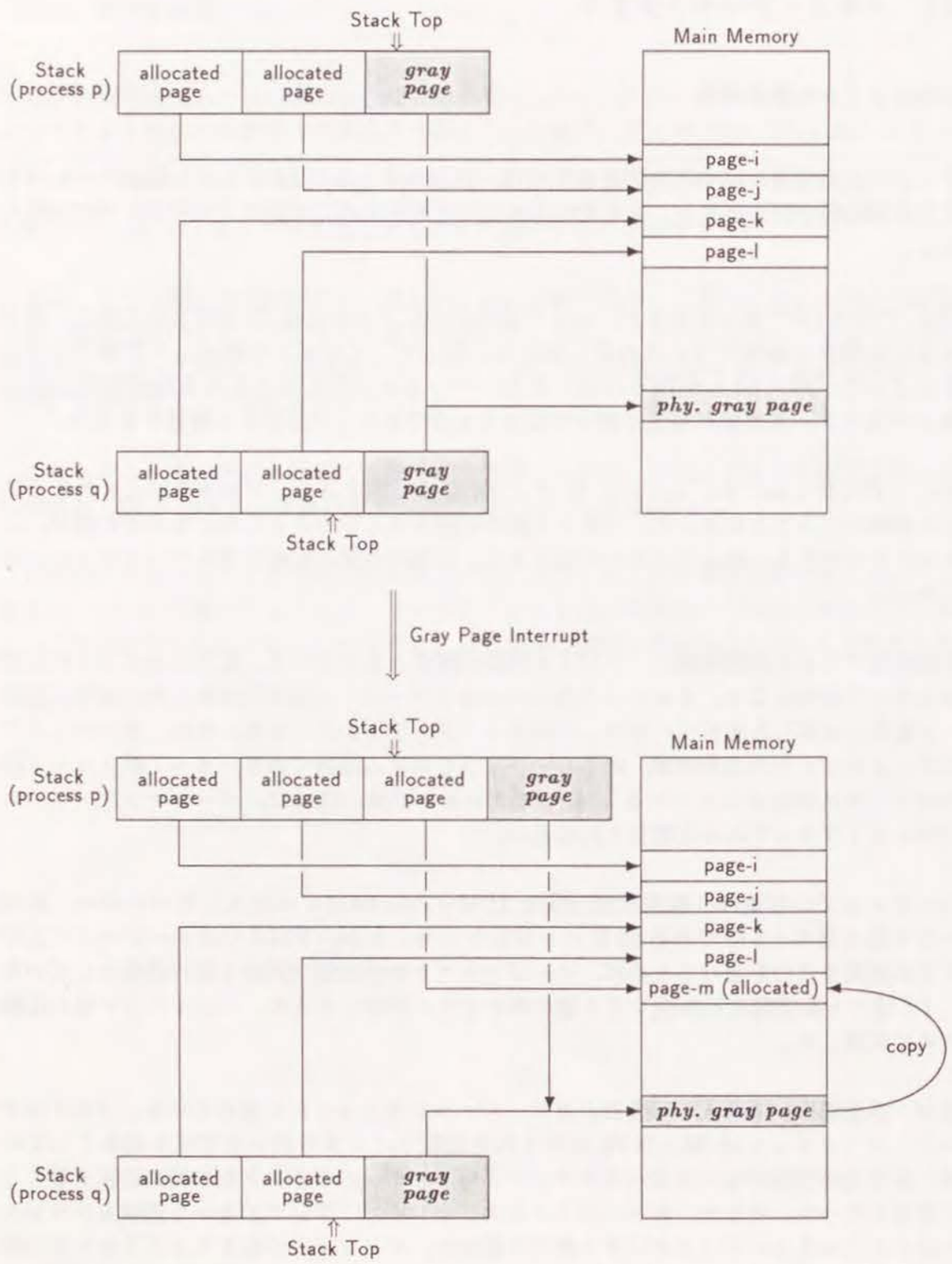


図 3-18: グレイ・ページ

一方グレイ・ページは，スタック／ヒープの伸長時のメモリ割当チェックを高速化するために導入された。PSI-Iでは3.1.6で述べたように特別なメモリ割当チェックはなく，マイクロプログラムによる判定を行っていたため，スタックを伸ばす操作のオーバヘッドとなっていた。そこで，PSI-IIでは各エリアに割当てられた最後のページを，グレイ・ページという特別なページとし，メモリ割当チェックのハードウェア化による高速化を図った[Nakashima 87c, Yoshida-H 86]。

グレイ・ページは，正しく物理ページが割当てられているが，アクセスを行うとアドレス変換割込が発生するという性質を持っている。従って，一つの機械命令によるスタック伸長を伴う書込は，グレイ・ページに対しても正しく行なわれ，命令の実行完了後に割込が発生する。この割込はマイクロプログラムによって処理され，新たな物理ページがグレイ・ページとして割当られ，それまでのグレイ・ページは「普通の」ページとなる（図3-18）。

この方式の問題点は，ほとんど使用されない物理ページがエリアの末尾に割当てられることによる，物理記憶空間の使用効率低下である。しかしこの問題は，スタックに関する物理グレイ・ページをプロセス共有とすることにより解決している。即ち，物理ページの割当処理において，それまでの論理グレイ・ページを L'_g，新たな論理グレイ・ページを L_g，物理グレイ・ページを P_g，獲得した物理ページを P とした時，それまでの論理／物理の対応関係；

$$L'_g \Rightarrow P_g$$

を，物理グレイ・ページを固定して；

$$L'_g \Rightarrow P$$
$$L_g \Rightarrow P_g$$

に変更する。このためには P_g から P へのコピーが必要となるが，一命令でのスタック伸長量は通常10w以下であるため，全く問題とはならない。また，マイクロプログラムの割込処理ルーチンはグレイ・ページ割込を最優先で処理するため，複数のプロセスによるグレイ・ページ書込みが競合する心配もない。一方，物理記憶の使用効率については，システム中に高々8ページの物理グレイ・ページが存在するだけであるので，8Mw（＝8Kページ）の最小構成システムであっても，無視できる大きさであると言える。

グレイ・ページに関するハードウェアは極めて簡単なものである。即ち，ページマップの各エントリには，グレイ・ページか否かを示すフラグが設けられており，アドレス変換のためのページマップの索引時に，グレイ・ページへのアクセスを容易に検知することができる。

グレイ・ページ方式の採用によって，ページ割当チェックのためのオーバヘッドは実質的に皆無となるが，その効果は以下のように見積もられる。まずコンパイラは一つの述語の

ヘッド・ユニフィケーションのために，各スタックが最大どの程度伸びるかを予測することができる*。また，ゴールとして現れるユニフィケーションによるグローバル／トレイル・スタックの伸びや，ゴール引数に現れる構造体生成のためのグローバル・スタックの伸びも予測できる。そこで，ゴール引数生成を行う *put* 系命令列の直前に；

- 呼出す述語のヘッド・ユニフィケーションでの伸び
- ゴール引数の構造体生成のための伸び
- 直前のゴール呼出し（またはヘッド・ユニフィケーション完了）の後で行われたユニフィケーション・ゴールによる伸び

をスタックごとに加えたものをオペランドとする命令；

```
check_stack_growth   dL, dG, dTR
```

を挿入することができる。この命令は一つのスタックに関して，スタック・トップと伸長量（dL など）の加算と，その結果とページ割当値の比較が必要であり，PSI-II では 7 ステップを要する。この値と述語呼出し命令の出現頻度 9%，及び一命令に要する平均ステップ数 5 を勘案すると；

$$\frac{5+7\times 9\%}{5}-1=12.6\%$$

が性能向上率であると推計される。これは非常に大きな値であり，グレイ・ページ導入の効果が極めて大きいことが確認された。

3.2.6.2 キャッシュ・メモリ

キャッシュ・メモリに関しては，ハードウェア量の削減のために容量を 4 Kw とし，かつダイレクト・マッピング方式としたことが最大の変更点である。この変更によって，ヒット率などの性能が低下することが当然予想されるが，その割合は 5.3.2 に示すように事前／事後のいずれの評価でも軽微であることが示されている。

また，多重論理空間の導入がキャッシュの構成にも影響を与えた。PSI-I では論理アドレスが物理アドレスに一対一に対応するため，キャッシュ・ディレクトリには論理アドレスを格納しておくことができた。しかし，PSI-II ではこの関係が崩れ，一つの論理アドレスが複数の物理アドレスに対応するようになったため，キャッシュ・ディレクトリに保持する値を変更する必要があった。

この問題の単純な解決法としては，キャッシュ・ディレクトリの内容を物理アドレスとすることが考えられるが，ヒット判定がアドレス変換後となるため，遅延時間の面で採用す

*トレイル・スタックに関しては構造体どうしのユニフィケーションがあるため完全な予測はできないが，予測が失敗することはまれであり，かつ容易に検知できる。

ることができなかった。また，プロセス ID と論理アドレスのペアとすることも考えられたが，ハードウェアによるプロセス数の制限や，ディレクトリのビット数が大きくなることを嫌って採用しなかった。

そこで，キャッシュ・ディレクトリが保持する値として，ページマップのアドレスを選択した。この値はスタックのグレイ・ページを除いて物理アドレスと一対一に対応し，またアドレス変換の途中結果であるため物理アドレスよりもかなり速い段階で確定する。問題はスタックのグレイ・ページであるが，これは前述のように物理ページ間でのコピーが行われるため，結果的には物理アドレスと一対一に対応することから，実際には問題とはならないことが明らかになった。

この他，キャッシュ・メモリの操作に関して，write-stack の改良と，cancel 操作の導入を行った。PSI-I の write-stack はアドレスに無関係に動作するため，ヒットすること若しくはキャッシュ・ラインの境界であることが保証されなければ使用できなかった。従って，スタック・フレームの先頭のワードに対しては通常の書込が必要であり，また 1 ワードのプッシュには使用できなかった。

そこで write-stack のアドレスがキャッシュ・ラインの境界でなければ，通常の書込とみなすという変更を行い，マイクロプログラムで自由に使用できるようにした。その結果 PSI-I では write と write-stack の比が 1 : 1.5 ～ 2.5 であったのが 1 : 3 に拡大し，使用頻度がかなり増加した。

一方 cancel 操作は，データ・タイプ判定とメモリ・アクセスの並列化のために導入された。例えば，ヒープ・ベクタの要素を更新する組込述語 `vector_element` は；

- 引数のデレファレンス
- 引数がヒープ・ベクタであることのチェック
- ベクタの先頭に置かれた要素数の読出

を最初に行うが，これは；

```
MAR = A[i] , if (A[i].tag == ref) { dereferece ; }
if (A[i].tag != hvect) { raise exception ; }
read(MAR, MDR) ;
```

という 3 ステップを要する。ここで，引数のタグが `hvect` であることを確認した「後で」読出を行っているのは，例えばアトムなどの値をアドレスとしてアクセスすることによるアドレス変換例外の発生や，キャッシュ・メモリの擾乱を防止するためである。

cancel 操作は，この確認と読出を並行して行うことを可能とするものであり，上記の例を；

```
MAR = A[i] , if (A[i].tag == ref) { dereferece ; }
read(MAR, MDR) , if (A[i].tag != hvect) { raise exception ; }
```

のように2ステップで実現する。即ち，"*raise exception*" の先頭で cancel が実行されると，直前に行われた読出によるアドレス変換例外やキャッシュのブロック・ロードなどが無効化される。従って上記の例のように，「普通は」アドレスであるが，「常に」そうとは限らないデータをアドレスとするメモリ・アクセスの効率化に寄与するものと期待できる。

3.2.7 マイクロ命令アーキテクチャ

PSI-I のマイクロ命令は，3.1.7 で述べたように使用頻度の低い機能に多くのビットが割当てられており，ある程度のビット数削減が可能であることが明らかになった。その一方で，マイクロ命令構成の不備によって，ハードウェア・レベルの並列性が充分に生かされていない点があることも指摘された。

そこで PSI-II では，PSI-I での評価や経験，また実行頻度が高いと思われる機械命令処理のマイクロプログラムのサンプル・コーディング結果に基き，マイクロ命令の設計を行った。その結果，図3-19に示すように，PSI-I よりも 10 ビット少ない 53 ビットのマイクロ命令で，ハードウェアの機能を最大限に発揮することができた。この命令長短縮とハードウェアの並列制御という相反する課題を解決するために大きな効果を発揮したのが，「エミット・フィールド」と呼ばれるマイクロ命令フィールド **EM1**, **EM2**, **EM3** である。これらのフィールドは，表3-5に示すように，即値や分岐アドレスの生成の他，使用頻度が比較的少ない機能のための命令コードとして用いられる。また，エミット・フィールドと機能の関係には綿密な配慮がなされており，使用頻度の高い機能が競合することが少ないように設計されている。

(1) TYPF

マイクロ命令のタイプを定めるフィールドであり，**CCF**, **DSTF**, **SC1F**, **SC2F**, **ALF** が持つ，使用頻度の高いタイプ1と，必須ではあるが頻度が低いタイプ2の，2種類の命令コードの選択を行う。この機能により，使用頻度の少ない操作のためにマイクロ命令フィールドが長くなるのを防ぐことができた。なお，分岐操作が重要であるという認識に基き，PSI-I のような分岐ができない命令タイプは設けないこととした。

また **TYPF** には，**EM1** を命令コード／即値として用いて，フラグ・レジスタ **MSTR** の制御，カウンタ **LC**, **G**, **S** の増減，及び DST-Bus のタグ部への即値出力を行う機能がある。これらの機能はいずれも使用頻度が比較的高いため，以下のような組合せで使用できるように配慮している。

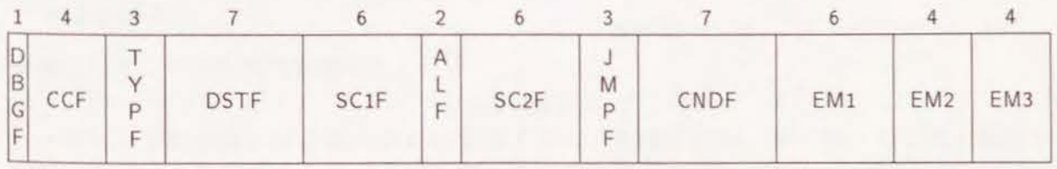


図 3-19: PSI-II のマイクロ命令

表 3-5: PSI-II のエミット・フィールド

機能		EM1	EM2	EM3
S1-Bus 定数生成	4 bit			○
	4 bit		○	
	6 bit	○		
S2-Bus 定数生成	4 bit			○
	4 bit		○	
	6 bit	○		
	8 bit with shift		○	○
	14 bit with shift	○	○	○
ALU 演算	special op.	(○)	○	
	hyper-special op.	(○)	○	○
分岐	eop, return			○
	short relative			○
	long relative		○	○
	unconditional absolute	○		○
	conditional absolute	○	○	○
タグ即値比較	long tag	○		
タグ即値生成	long tag	○		
	short with counter op.	○		
フラグ制御	general	○		
	special	○	○	

(1) ALU ステータス・ロード（EM1 不要）
(2) ALU ステータス・ロード＋カウンタ制御＋タグ即値（限定）
(3) ALU ステータス・ロード＋タグ即値（任意）
(4) スイッチ・フラグ制御＋カウンタ制御

評価の結果では，これらのいずれかが使用される頻度は 50% を超えており，細心の注意を払った設計が正しかったことが立証された。また，カウンタ制御の実行頻度は **LC** を主に 19% と高く，導入の効果が良く現れている。

(2) DBGF

PSI-I と同様，ブレーク・ポイントの設定と評価用カウンタ **GEVC** のインクリメントを行う。但し，どちらを実行するかを制御フラグの値により定めることとし，PSI-I での 2 ビットを 1 ビットに削減した。

(3) CCF

メモリ・アクセス操作と，アドレス／データ・レジスタの選択，及びアドレス・レジスタのインクリメント制御を行う。PSI-I ではこれらの制御は別々のフィールドで指定されていたが，使用頻度が少ないと思われる組合せを排除し，ビット数の削減を図った。また，命令タイプの導入や極端に使用頻度が少ない操作のアドレスを用いた選択などにより，PSI-I での 7 ビットを 4 ビットにまで圧縮することができた。

(4) DSTF

DST-Bus の値を書込むレジスタを選択するフィールドであり，コードの大半は **RF** の直接アクセスのために用いられている。PSI-I で有用性が確かめられた同時書込機能は，**RF** の直接アクセス領域と **MDR/MAR** を対象としたものは踏襲した他；

- **MDR+MAR**
- **IFR+MAR**
- **LC+MDR**
- **G+MAR**
- **S+MAR**

を導入して，一層の充実を図った。

一方ビット数削減の面では，**RF** の直接アクセス領域の圧縮，タグ部への「値」書込を **MDR** のみとしたこと，命令タイプの導入などが効果を発揮し，PSI-I での 10 ビットが 7 ビットとなった。また，PSI-I では独立したフィールドであった I/O バス制御を，**DSTF**（タイプ 2）の機能としたことも，全体のビット数削減に寄与した。

(5) SC1F

S1-Bus に出力するレジスタを選択するフィールドであり，コードの半分は **RF** の直接アクセス領域のアドレス生成に用いられている。この領域が圧縮されたことや，命令タイプ導入によって，フィールド長は PSI-I よりも 1 ビット減少している。また，エミット・フィールドを即値とする定数生成を用意し，定数領域の廃止に対処している。

(6) SC2F

S2-Bus に出力するレジスタを選択するフィールドであり，PSI-I から大きく変更されたものの一つである。まず，S2-Bus へ出力できる **RF** の直接アクセス領域が，PSI-I の 16 w から 32 w へ増加し，更に間接アクセス領域の出力も可能となった。また定数領域の廃止に対処するために，S2-Bus に対する定数出力のソースとして，個々のエミット・フィールド及び複数を連結したもの，更にはそれらをバイト単位に巡回シフトしたものを用意した。これらに加え，**IR** などのレジスタのバイト単位のシフト／マスク操作の指定も，**SC2F** で行うこととした。

これらは全てフィールド長を増やす方向の改良であるが，実現機能の綿密な選択や命令タイプの導入により，PSI-I と同じビット数にエンコードすることができた。なお，PSI-I ではシフト／マスク操作は別のフィールドで制御されており，実質的には 4 ビットの削減となっている。

(7) ALF

ALU で行う演算の種類を定めるフィールドである。演算チップ Am29332 は 7 ビットの命令コードを持ち，128 種類の演算を行うことができるが，これらの内 PSI-I の評価結果で支配的であった加減算と AND 演算のみを **ALF** で直接制御することとした。これら以外については，OR などの論理演算，シフト，乗除算サポートなど，比較的高頻度と思われる 16 種をタイプ 1 の演算として，**EM2** により選択することとした。それ以外はタイプ 2 の演算とし，Am29332 のコードを直接 **EM2** と **EM3** で指定することとした。

この他，24 ビット系の演算を削除するなどした結果，**ALF** は 2 ビットとなり，PSI-I に比べて 1 ビット削減することができた。なお，シフトの桁数は **EM1** またはレジスタ **LC** により与えることとし，PSI-I の **BIRF** は廃止した。

ALU 演算の頻度を測定した結果 [Yoshida-H 87]，まず全くデータ転送／演算を行わないものは約 10% と，PSI-I に比べて半減した。これは，分岐操作を中心としたマイクロ命令構成の改良の効果であると考えられる。なお，ALU を全く使用しない命令の約 2/3 が，分岐とメモリ・アクセスのみを行っているものである。

一方，`Y = X + 0` のような単なるデータ転送は，PSI-I の 40% から 56% に増加した。ハードウェア構成上，このような転送を特に高速化することはできなかったが，一考に値する結果ではある。また，演算を行っているものについては，その約 85% が加減算であり，予想が裏付けられた。

(8) JMPF/CNDF

分岐タイプの指定 (**JMPF**) と，分岐条件の指定 (**CNDF**) を行うフィールドであり，PSI-I から大きく変更されたものの一つである。

まず，分岐と条件の指定を分離したことにより，条件付きの eop，サブルーチン呼出／復帰が可能となった。この操作は 3.2.4 で述べたように，特に eop に対して有効であると考えられる。

分岐アドレスの生成については；

- **EM3** のみを用いた自己相対アドレス
- **EM2** と **EM3** を連結した自己相対アドレス
- **EM1**, **EM2**, **EM3** の全てを連結した絶対アドレス

の三種を用意し，頻度の高い近傍への分岐については，他の機能のためにエミット・フィールドを有効に使用できるようにした。また，条件付きの絶対アドレス分岐の導入は，マイクロプログラムの書き易さの面で効果が大きいものと考えられる。更に，**CNDF** が不要である無条件分岐については，**EM2** のかわりに **CNDF** を絶対アドレス生成に使用し，S1/S2-Bus への定数生成などの機能を同時に実行できるようにした。

一方，マイクロ命令長削減の面では，eop やサブルーチンからの復帰など，アドレスをマイクロ命令中に必要としないものの選択を，**EM3** を用いて行うようにした。

CNDF に関しては，タグ即値との比較条件を中心に強化した。即ち S1-Bus のタグ部と即値との一致／不一致のどちらも分岐条件にできるようにした他，上位ビットのみを対象としたマスク付き比較も導入した。また，タグ即値の指定方法については，使用頻度の高い 0 ～ 15 については **CNDF** の中で直接に，またそれ以外については **EM1** を使用することとし，機能とマイクロ命令長の双方に配慮した構成とした。

その他の分岐条件については，**MSTR** を 32 ビットから 24 ビットに縮小した他，タグ／値の双方が等しいことを示す "`same_tag && zero`" や，アドレス比較のためのキャリーとゼロの組合せ，更には数値比較のための符号とオーバフローの Exclusive-OR など，様々な複合条件を導入した。

分岐に関する評価の結果では，PSI-I と同様に約 70% の命令が何らかの分岐を行ってい

ること，その中の約70%が二方向条件分岐であることが示され，分岐操作の重要性が再確認された。

3.3 パイプライン方式による逐次型推論マシン (PSI-III)

3.3.1 設計方針

前節で述べたように，PSI-IIではWAMをベースとした機械命令の導入，ハードウェア・アーキテクチャの様々な改良などによって，PSI-Iを大幅に上回る高い性能を達成することができた。また，LSI化やアーキテクチャの工夫により，PSI-Iの1/4というコンパクトな実装を行うことができ，筐体の小型化や64プロセッサからなるMulti-PSI/v2の実現が可能となった。

しかし，PSI-IIをもってしても多大な実行時間を要する大規模アプリケーションは数多く存在し，一層の性能向上に対する期待や要望は依然として強いものがあった。また，第五世代コンピュータ・プロジェクトの最終成果の一つである並列推論マシンPIM [Uchida 88, Goto 88b] は，数百プロセッサ規模の大規模並列処理を目標としており，単位プロセッサの更なる小型化が求められていた。

そこでまず，PSI-IIをより集積度の高いVLSIチップを用いて再構成し，実装規模の圧縮とともに，高集積ゲートの使用によるマシン・サイクルの短縮を行うことを検討した。しかしこのような単純な改良では，マシン・サイクルは100 ns程度にしか縮まらず，1.5倍程度の性能向上に留まることが明かとなった。この理由は，PSI-IIのクリティカル・パス上のレジスタ・ファイル，ALUなどの素子には，内部がECLで構成されたAm29300を使っているため，1 μm ないしはサブミクロン・プロセスのCMOSゲートを用いても大きな遅延時間短縮が望めないことにあった。また実装の面では，大容量のSRAMを中心としたハードウェア化されたアドレス変換機構の存在がネックとなることが判明した。

そこで高速化を主眼に，PSI-IIの機械命令の実行過程や頻度を詳しく検討した結果，以下のことが明らかになった [Nakajima 87]。まず，実行時間の約1/3はEnvironment/Choice Pointの生成と，バックトラックによるChoice Pointからレジスタへの移動に占められているが，これらは；

RF ⇔ MDR ⇔ メモリ

の転送が主である。この内**RF ⇒ MDR**の転送は，クリティカル・パスの一つである；

RF ⇒ ALU ⇒ RF

よりもかなり短い時間で実現可能である。また3.2.7で述べたように，ALU演算が行われないマイクロ命令が約2/3を占めていることもあり；

(a) **RF ⇒ MDR**

(b) **MDR** ⇒ ALU ⇒ **MDR/RF**

の二つに分割する「2フェーズ」構成が有望であると考えられた。これらのパスの遅延はどちらも60 ns程度と見積もられ*，同じ操作に関する限りPSI-IIの2.5倍の性能が得られることが判明した。

一方，やはり実行時間の約1/3を占める *get, put, unify* 系，及び組込述語の実行に関しては，**RF** をソースとするALU演算や，**RF** から **RF** の転送が2サイクル要することは，かなりのダメージとなる。これを回避するには，単純な2フェーズ構成ではなく，命令実行開始時にはオペランドが **RF** から **MDR** に移されている，「2ステージ・パイプライン」の導入が必要であることが判明した。またこの機能は，`get_variable Yn,Ai` のように，**RF** の内容をメモリに書込む命令の高速化にも役立つことが判った。

さて，PSI-IIのもう一つの重要なクリティカル・パスは；

RF ⇒ タグ即値比較 ⇒ 分岐条件選択 ⇒ 分岐アドレス生成 ⇒ マイクロ命令読出

であるが，マシン・サイクルを60 nsとするためには；

(a) **RF** ⇒ タグ即値比較 ⇒ 分岐条件選択
(b) 分岐条件 ⇒ 分岐アドレス生成 ⇒ マイクロ命令読出

の分割がやはり必要となる。従って，単純な転送を除く *get/unify* 系命令や組込述語の処理のためには，**RF** ⇒ **MDR** の転送だけではなく，タグ判定のパイプライン化が必要であることが判った。

また**MDR**のタグ判定に関してもPSI-IIと同じ構成では60 nsをかなり上回ることが見積もられ，**RF** と同様に分割せざるをえないことが判明した。しかしこの分割は，特にデレファレンスに大きな悪影響を及ぼすため，何らかの対応を必要とすると考えられた。そこで，Reference Pointerの連鎖の長さが0であることが約2/3，1であることが約1/3，2以上であることは極めてまれであることに基き；

- デレファレンスのルートである **RF** の読出とタグ判定
- 判定結果に基くメモリの読出

のパイプライン化を考えた。この機能は通常のパイプライン・プロセッサにおける，アドレス計算とオペランド・フェッチに相当し，アドレス計算機能を充実すれば `put_value Yn,Ai` のように単にメモリを読出して **RF** に書込む命令の高速化にもつながる。また，`get_variable Yn,Ai` におけるアドレス計算や，`execute` のような無条件分岐の高速化も可能である。

*RF を読出す操作はそのアドレスの生成がクリティカル・パスに含まれるため，アドレス生成が並行に行われる書込操作に比べて遅延時間が大きくなる

しかしこのような構成では，メモリ上のオペランドについては遅延時間の関係で，最終パイプライン・ステージにデータを渡すだけしか行えず，前述のようなタグ判定のパイプライン化はできない。従って，最終ステージでオペランドのタグ判定がもう一度必要となり，場合によってはデレファレンスも続行しなければならない。また，KL1ではReference Pointerの連鎖がKL0に比べて長くなる傾向にあるため，連鎖の長さについての前提が崩れることも考えられる。

そこで，メモリ上のオペランドのタグ判定を行うとともに，デレファレンスを完全に終了させるためのステージを，メモリ読出ステージと最終ステージの間に挿入することとした。これにより，データ・タイプ判定が完全にパイプライン化されるとともに，判定結果によって命令処理のためのマイクロプログラム・ルーチンの実行開始番地を変える，条件分岐のパイプライン化も可能となった。

以上をまとめると，PSI-IIIのパイプライン構成は以下のようなものとなる [Nakashima 90c, 90d]。

A アドレス計算
R メモリ・オペランド・フェッチ
S データ・タイプ判定
E 演算

更に **A** ステージの前にデコード用のステージ **D** を加え，5ステージのパイプラインとすることとした。

このようなパイプライン方式を導入したことは，メモリ・アーキテクチャにも影響を及ぼした。即ち5.3.2で述べるように，PSI-IIのメモリ・アクセス頻度は60%以上と極めて高く，またその2/3がデータ・アクセスに占められている。従って，メモリ・アクセスのパイプライン化によりアクセス頻度がさらに増加すると，命令フェッチとの競合が重大な問題となることが予想される。

そこで，命令キャッシュとデータ・キャッシュを分離して，並行にアクセスすることを可能とするハーバード・アーキテクチャを採用することとした。また実装上の問題から，ハードウェアのアドレス変換機構を廃止し，所謂TLB (Table Look-aside Buffer) に置き換えたのも大きな変更点である。

この他，三大基本方針の一つであるマイクロプログラミングに関しては，PSI-IやPSI-IIで証明されたその有効性を生かすために，**E** ステージをマイクロプログラム制御とすることとした。更に，命令デコードにナノ・プログラミングの考え方を導入して，開発の効率化と柔軟性の保持を図り，PSI-IIとのオブジェクト・コード・レベルでの互換性を保つことができた。

これらの結果，PSI-Ⅲの実行速度はPSI-Ⅱの2～3.5倍となり，1.5 MLIPSという極めて高い性能を達成することができた。また，CPUの実装規模はPSI-Ⅱの1/3となり，256プロセッサからなる大規模並列推論マシンPIM/mの実現が可能となった。

3.3.2 ハードウェア構成

PSI-Ⅲのハードウェアは，図3-20に示す構成となっており，マシン・サイクルは60 nsである。

主記憶（物理記憶）は従来機と同様であるが，最大実装容量は64 Mw，また物理アドレス幅は32ビットに拡張されている。

アドレス変換機構は前述のようにTLBに置き換えられ，命令／データ用それぞれ64エントリ，セット・アソシアティブ方式（2ウェイ）のものが備えられている。また，命令キャッシュ／データ・キャッシュの容量はそれぞれ1 Kw，4 Kwであり，共にダイレクト・マッピング方式である。この他，バックトラックの高速化のために，16 wのトレイル・バッファが新たに設けられた。これらのハードウェアはデータ・キャッシュのデータ・アレイを除き，全てVLSIチップCU上に実装されている。CUに搭載されたトランジスタ数は約610 Kであり，1μmのCMOSプロセスにより製造される。なお，TLBの構成に関

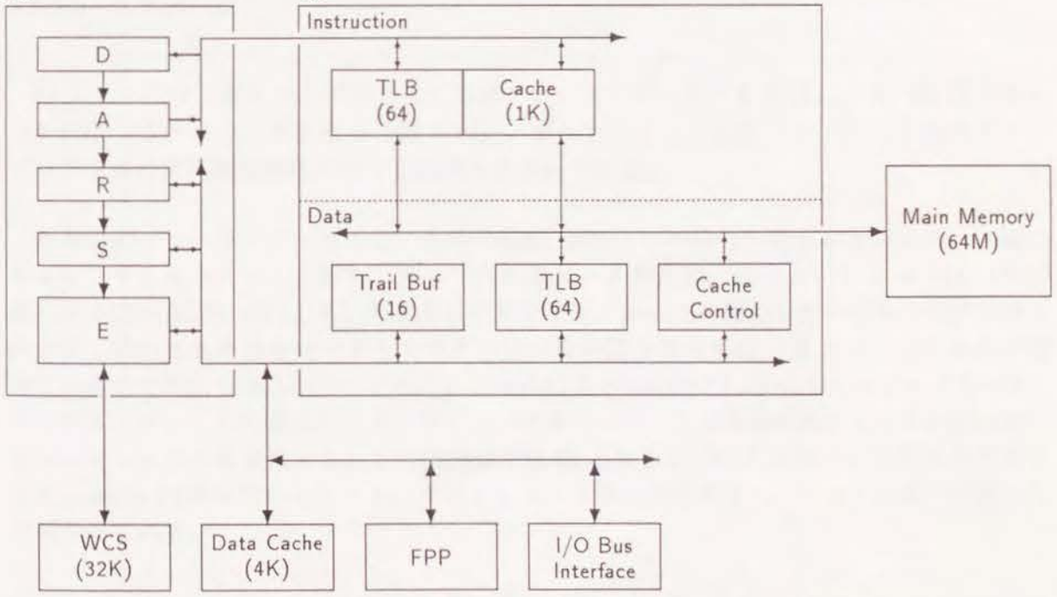


図3-20: PSI-Ⅲのハードウェア構成

しては3.3.5で，トレイル・バッファについては4.2.2で，それぞれ詳しく論ずる。

プロセッサに関しては，前述の5ステージ・パイプラインの全てがVLSIチップPU上に実装されている。PUの搭載トランジスタ数は約380Kであり，製造は0.8 μmのCMOSプロセスを用いて行った。また，WCSはチップ外部に置かれた高速SRAMで構成されており，64 bit × 32 Kwの容量を持っている。この他，I/Oバスの制御などを行うためのLSIや浮動小数点演算プロセッサが備えられており，これら全てが1枚のプリント基板上に実装されている。従って，CPUの実装規模はPSI-IIの1/3に圧縮されている。

3.3.3 パイプライン・アーキテクチャ

前述のようにPSI-IIIのプロセッサ・チップであるPUは；

- **D** デコード
- **A** アドレス計算
- **R** メモリ・オペランド・フェッチ
- **S** データ・タイプ判定
- **E** 演算

の5つのパイプライン・ステージから構成されている（図3-21）。これらの内**A**, **R**, **S**の三つのステージは**D**ステージが生成する「ナノコード」により，また**E**ステージはマイクロプログラムにより，それぞれ制御される。また，**E**ステージは，**RF**などのレジスタからメモリ・アドレス／データ・レジスタに転送するフェーズ1と，メモリ・アドレス／データ・レジスタの値をソースとする演算を行うフェーズ2に分かれており，これらは並行動作する。なお，フェーズ1は**S**ステージと共有されており，オペランドのセット・アップに用いられる。

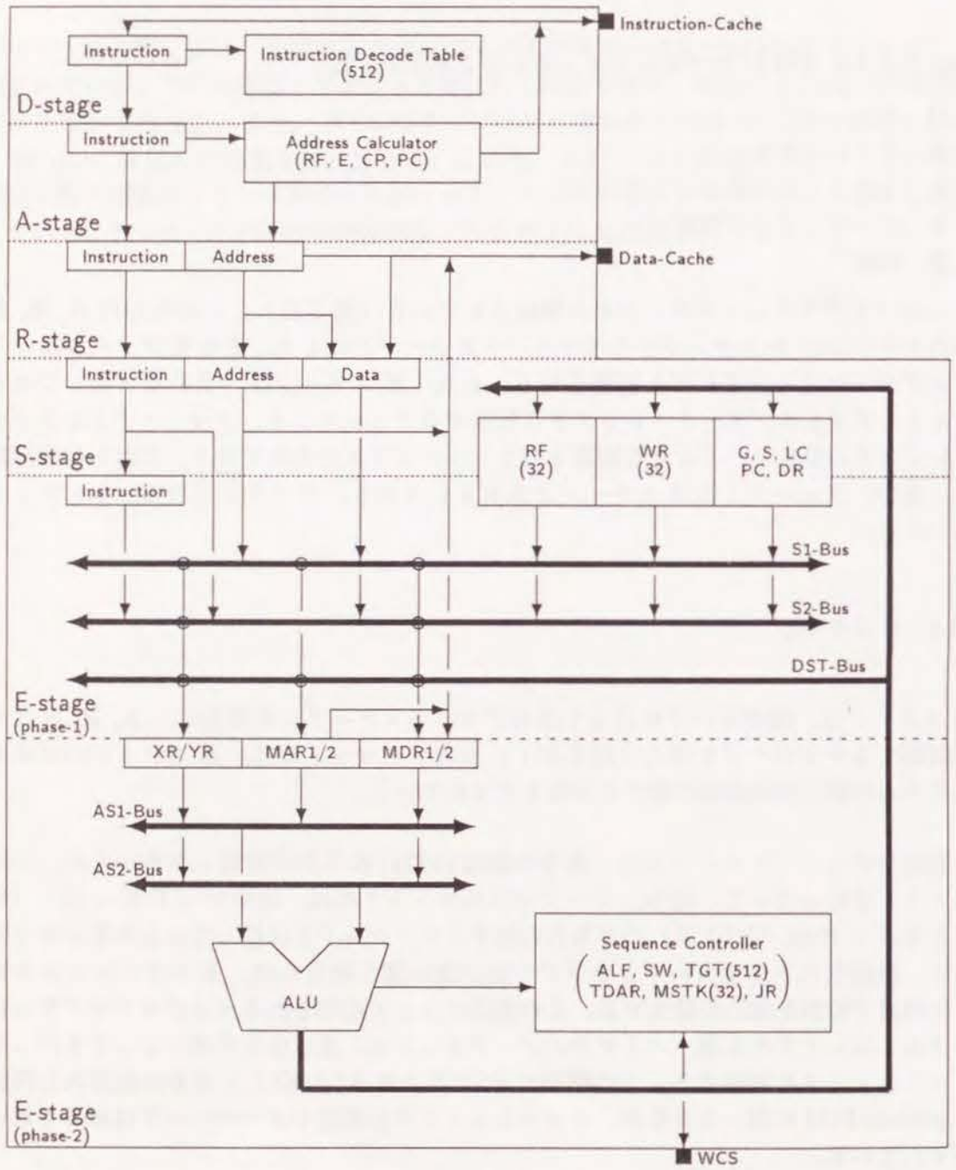
3.3.3.1 Dステージ

Dステージは，命令コードによって命令デコード・テーブルを索引し，**A**, **R**, **S**ステージを制御するナノコードを得る（図3-22）。またナノコードには，**E**ステージのマイクロプログラムの実行開始番地に関する情報も含まれている。

なお命令デコード・テーブルは，命令の種類(1024)の半分の容量しかないため，一種のキャッシュとなっている。即ち，テーブルの各エントリには，命令コード0 ～ 511（KL0系）と512 ～ 1024（KL1系）のどちらに対するナノコードを保持しているかを示すフラグがあり，供給された命令コードとフラグの値が食い違う場合には，**E**ステージに命令が到達した時点で特別な割込が発生する。この割込によって起動されるマイクロプログラムは，テーブル・エントリの置換，パイプライン・フラッシュ，及び命令の再フェッチを行って，キャッシュ・ミスに対処する。この機能はPSI-III上にKL0/KL1の双方の処理系を同居させるpseudo-PIMに用いられるが，キャッシュ・ミスの頻度やオーバヘッドは極めて低いと予想されている。

3.3.3.2 Aステージ

Aステージは，ナノコード・フィールド**ACF**に従って，以下のレジスタをソースとするアドレス計算を行う。



D-stage: Decode Stage
A-stage: Address Calculation Stage
R-stage: Read Data Stage
S-stage: Set-up Stage
E-stage: Execution Stage

RF: Register File
WR: Work Register File
E: Environmen Base
CP: Continuation Point

PC: Program Counter
G: Global Stack Top
S: Structure Pointer
LC: Loop Counter
DR: Data Reg

ALF: ALU Flags
SW: Switch Flags
TGT: Tag Dispatch Table
TDAR: Tag Dispatch Addr. Reg.
MSTK: Microprogram Stack
JR: Jump Reg.
WCS: Writable Control Store

図 3-21: PSI-III のパイプライン構成

3	1	2	4	2	2	3	3	3	2	2	3	4	6	10
ACF	ASF	DFF	LKF	IFF	LNF	ODPF	OS1F	OS1F	ODSF	MDF	TDPF	EMO1	EMO2	MAF

図 3-22: ナノコード

RF 引数レジスタ A_i のための 32 w のレジスタ・ファイルであり，そのアドレスは命令のフィールド `r0` ～ `r2` が供給する。またデレファレンスのルートとなるため，アドレス計算に用いられたエントリのタグが `ref` であるか否かの判定も行われる。

E Environment のベースであり，局所変数のアドレス計算に用いられる。

CP 復帰アドレスを保持し，`proceed` の分岐アドレス生成に用いられる。

PC プログラム・カウンタであり，自己相対分岐アドレスの計算に用いられる。

なお，文字列型データなどのバイト・アクセスのために，生成したアドレスを 2 ビット右シフトする機能もあり，ナノコード・フィールド **ASF** で制御される。

上記の内 **PC** 以外については，その値の設定は **E** ステージが行うため，インタロック制御が成される。即ち，**A** ステージには **RF** の各エントリと **E**, **CP** に対応するインタロック・フラグがあり，これらのレジスタを更新する命令はナノコード・フィールド **LKF** によりインタロックを行う。

PC に関してはインタロック機構はなく，分岐を含む命令フェッチを **A** ステージが行ってその値を決定する。このため，ナノコードにはフィールド **IFF** があって，分岐を行うか否かが指定される。この他，命令長が 1 w/2 w のいずれであるかを示すフィールド **LNF** も，**A** ステージの制御に用いられる。

3.3.3.3 R ステージ

R ステージはナノコード・フィールド **DFF** の指示と，**A** ステージが生成したアドレスに従って，メモリ・オペランドをフェッチする。オペランド・フェッチには以下の三種類がある。

(1) 局所変数の読出など，デレファレンスは行わない単なるフェッチ。

(2) **RF** をルートとするデレファレンス。タグが `ref` でなければフェッチしない。

(3) 局所変数などメモリ・オペランドをルートとするデレファレンス。**R** ステージでは無条件にフェッチし，以後のデレファレンスは **S** ステージが行う。

3.3.3.4 S ステージ

S ステージでは，デレファレンスを含むオペランド・セットアップと，データ・タイプ判定を伴うマイクロプログラム実行開始アドレス生成が行われる。これらの内，デレファレンスとデータ・タイプ判定については 3.3.4 で詳しく論ずる。

オペランド・セットアップは，**E** ステージが持つ二つのメモリ・アドレス・レジスタ **MAR1**/**MAR2** と，メモリ・データ・レジスタ **MDR1**/**MDR2** に対して，命令のオペランドなどを設定する操作である。セットアップ操作のソースは，以下のものから選択される。

(1) メモリ・オペランド及びそのアドレス
(2) 命令のオペランド・フィールドを符号／ゼロ拡張を施して抽出した値
(3) 命令フィールド `r0` ～ `r3` で定まる **RF** のエントリ
(4) 制御レジスタ / スクラッチ・パッドのためのレジスタ・ファイル **WR**
(5) グローバル・スタック・トップ **G**
(6) 構造体ポインタ **S**

これらの内の三つが，ナノコード・フィールド **OS1F**, **OS2F** により選択され，**ODSF** が定めるレジスタに設定される。また，**OTF** の指示によりエミット・フィールド **EMO2** をタグ即値として付加することもできる。

なお，(4) ～ (6) は所謂オペランドではないが，パイプラインによる並行動作を最大限に活用するために，セットアップの対象とした。例えば `deallocate` 命令にはオペランドはないが，**WR** のエントリである **E** と **B** をセットアップすることにより，Environment 除去の可否を判定するための **E** と **B** の比較演算を即座に行うことができる。また，**G** と **S** は *unify* 系の命令で用いられるため，読出／書込のどちらのモードかを示すフラグ `MODE_A` の値によってソースを選択する機能も備えられている。

さて，上記のレジスタの内 (3) ～ (6) は **E** ステージと共有のレジスタである。更に，セットアップを行うためのバス，フィールド抽出機構，レジスタ入力選択回路など，データ系の回路全てを **E** ステージと共有とすることにより，セットアップに必要なハードウェアを最小限のものとしている。即ちこれら **E** ステージのフェーズ 1 回路は，通常はマイクロ命

令によって制御されるが，`eop` 操作が発効される場合*には，**S** ステージの制御下に置かれる。これは，抽象機械の状態を定めるレジスタ類がフェーズ 2 には無いため，`eop` 操作と同時に行われるフェーズ 1 からフェーズ 2 への転送は無意味であるという事実に基いている。なお，`eop` が条件付きのものであれば，もちろん分岐条件が真の時にのみセットアップが行われる。また，`eop` と同時に行われるフェーズ 1 レジスタへの更新操作については，元来必要な ALU からフェーズ 2 レジスタへの書込パスなどを利用してバイパスされる。

マイクロ命令の実行開始アドレスの生成は，ナノコード・フィールド **ODPF** によって制御される。アドレスは原則的には命令コードと **MAF** によって定められるが，上位ビットを **EMO2** で指定することや，下位ビットを様々な条件によって修飾することによる二方向／多方向分岐も用意されている。分岐条件には，3.3.4 で述べるタグ判定結果によるものの他，*unify* 系の命令の読出／書込モードによるもの，4.2.2 で述べるシャロー・バックトラックの最適化に関する実行モードによるもの，及びこれらを組み合わせたものが用意されている。これらの実行モード・フラグは，**MDF** の制御によりオン／オフすることができる他，読出／書込モード・フラグをタグ判定の結果で設定する機能も備えられている。

この他，ナノコード・フィールド **TDPF** を用いて，**E** ステージで行うタグ多方向分岐のセットアップを行う機能もあるが，これについては 3.3.4 で述べる。

3.3.3.5 E ステージ

E ステージは前述のように，並行に動作するフェーズ 1 とフェーズ 2 に分かれている。フェーズ 1 には前述の **RF**, **WR**, **G**, **S** の他，ループ・カウンタ **LC**，プログラム・カウンタ **PC**，命令レジスタ **IR**，及び順序制御系や他のステージのレジスタをアクセスする際のインタフェースとなるレジスタ **DR** がある。フェーズ 1 からフェーズ 2 へは，S1-Bus/S2-Bus を経由して，同時に二つのレジスタの値を転送することができる。S1-Bus には **IR** を除く全てのレジスタが，S2-Bus には **RF**, **WR**, **IR**，及び 2 ワード命令のために **S** ステージの命令レジスタが，それぞれ接続されている。このように二つのバスによるレジスタ転送により，PSI-II で高い頻度で行われていたレジスタ転送を並列処理することが可能となった。これに加え，フェーズ 2 からフェーズ 1 への転送も並行して行うことができるため，2 フェーズ化による処理ステップ数の増加を最小限とするばかりか，かえってステップ数の削減につながることも期待される。

また，S1-Bus/S2-Bus とも定数出力機能があり，特に S2-Bus に関しては PSI-II と同様のエミット・フィールドの連結やそれをシフトした値を定数として出力することができる。更に，**IR** に関してはフィールド抽出や **RF** のアドレス生成の機能があり，**S** ステージによるオペランド・セットアップを補完している。

*) **E** ステージがアイドルの場合も含む。

フェーズ2には，前述のようにメモリとのインタフェースとなるMAR1/MAR2及びMDR1/MDR2の他に，スクラッチ・パッドとしてもちられるレジスタXRとYRがある。これらの内，MAR1とMAR2はALUの入力バスの一つであるAS1-Busのみに，それ以外についてはAS1-Busともう一つの入力バスAS2-Busの双方に，それぞれ接続されている。また，AS2-Busに定数を出力する機構も備えられている。

ALUでは加減算，論理演算の他，バイト単位のフィールド抽出演算，シフト演算が行われ，演算結果はDST-Busを経由してフェーズ1／フェーズ2の双方のレジスタに書込むことができる。なお，乗除算は浮動小数点演算プロセッサを用いて高速に行うため，ALUにはそれに関する機能はない。

順序制御系については基本的にPSI-IIと同様であるが，前述のように分岐条件の生成が2フェーズに分割されたことが大きな変更点である。即ち，タグと即値の比較結果，ALU演算結果を保持するフラグやスイッチ・フラグなどから分岐条件が選択されて，一旦分岐条件フラグCONDに設定され，その次のステップでCONDに基く条件分岐が行われる。この変更は，条件分岐の頻度が極めて高いことを考えると，かなりのダメージをもたらすものと予想された。そこで，タグ判定とともに頻繁に分岐条件として使用されるALU演算結果に関しては，ステータス・フラグを経由せずに直接CONDに設定できるようにした。この結果，ALU演算結果に関してはPSI-IIと同様の条件分岐が可能となり，タグ判定のパイプライン化と合わせると，ダメージはかなり小さくなったものと考えられる。

この他，インタロック，パイプライン・フラッシュなどのパイプライン制御や，3.2.5で述べるTLBミスの処理のためのマイクロ命令レベルのトラップが，Eステージの機能として新たに加えられた。また，従来はマイクロプログラムによるレジスタ・アクセスのレベルで実現されていた，I/Oバス制御やタイマーなどの機能は，メモリ・アクセスと同様のインタフェースとして，PUの外部に実装することとした。

3.3.3.6 パイプライン制御

命令パイプラインを設計する上で，データの依存関係を解決するためのインタロック制御と，分岐命令の制御が重要な項目であるが，これらに関しては以下のような構成とした。

まずインタロックに関しては，論理型言語の場合にはゴールを呼び出す際の引数レジスタの設定と，呼び出された述語のヘッド・ユニフィケーションでの引数の参照が，データ依存の基本的な関係となっている。従って，RISCなどのようにインタロックがないように命令の順序などを調整することは，依存関係が複数の述語にまたがっているため容易ではない。またPSI-IIIのオブジェクト・コードは，PSI-IIのそれと互換であることが要求されていたので，命令の順序調整などを行うことは，実際問題として不可能であった。

そこで，3.3.3.2で述べたように，Eステージによる引数レジスタの更新と，Aステージによるデレファレンスのための参照が前後しないようにするためのインタロック機構を設けて，引数レジスタの依存関係を解決することとした。この機構によりデレファレンスを行う命令は，先行する引数レジスタの更新命令が完了するまでAステージに留まるが，それによるオーバヘッドはパイプラインの段数に大きく影響される。即ちPSI-IIIでは3.3.4で述べるように，タグ判定とデレファレンスのパイプライン化のための特別なステージであるSステージがあり，その結果Aステージと Eステージの「距離」が遠くなっているため，オーバヘッドが大きくなることが懸念される。

しかし，PSI-IIIにおける論理型言語の処理方式の性質上，この問題はさほど重大なものではないと予想された。即ち，n引数の述語に関するi番目の引数$\mathbf{A_i}$の設定とその参照の間には；

(1) $\mathbf{A_{i+1}}$～$\mathbf{A_n}$の設定
(2) ゴールの呼び出し
(3) Choice Point生成など，クローズ選択のための処理
(4) $\mathbf{A_1}$～$\mathbf{A_{i-1}}$に関するヘッド・ユニフィケーション

があり，多くの場合更新と参照を行う命令間に，充分な数の命令または処理サイクルが存在することが期待できる。例えば*append*の場合，引数が3個と少なく，かつClause Indexingのために(3)の処理で第一引数を参照しているが，設定と参照の間に5命令*が存在し，パイプライン・ストールは発生しない。

一方，メモリ・データに関する依存関係の解決は，メモリの書込アドレスとメモリ・オペランドのアドレスとの一致検出機構をSステージに設けることにより行っている。この機構は，論理型言語の処理ではメモリ・アクセスの頻度が高いことを考慮して設けられた。即ち，汎用のマイクロプロセッサの中には[Yoshida-T 90]のように，メモリ書込命令と読出命令がインタロックするような物もあるが，5.3.2で述べるようにWAMでは一命令あたり約二回のメモリ・アクセスが行われるため，パイプライン・ストールが頻発すると考えた。なお，書込／読出アドレスの一致が検出されると，パイプライン・フラッシュが行われるが，このような状況は同じ変数に対するユニフィケーション命令が連続する場合に限られるため，極めて稀であることが予想される。また3.3.4.3で述べるように，パイプラインによるデレファレンスの後で，Reference PointerをEステージが書き換えることがないことは，論理型言語の持つ単一代入の性質により保証できる。

分岐命令に関しては，3.3.3.2で述べたように，Aステージで分岐アドレスの計算とフェッチを行うが，条件分岐に関する機構は特に用意されていない。即ち，条件分岐命令の実行時に，Aステージが分岐側をフェッチするか，非分岐側をフェッチするかは，命令によってあらかじめ定められており，その「予測」が外れた場合にはEステージがパイプライン・

*4.4で述べる複合命令を用いると4命令。

フラッシュと再フェッチを行う。従って，分岐条件の動的な予測機構，複数の命令ストリーム，遅延スロットなど，条件分岐を高速化するための機構は特に設けられていない。

このような単純な設計とした理由は，論理型言語処理における条件分岐のほとんどがユニフィケーションや組込述語の「失敗」によるものであること，つまり「予測していない」側はバックトラックを行うことにある。即ち，バックトラック処理では Choice Point に退避された情報の復元や Undo のような，時間のかかる操作が行われるため，これらを実行する前に再フェッチを開始すれば，バックトラック処理の完了時には候補節のための命令列がパイプラインに充填されていることが期待できる。実際，通常のバックトラック処理では，Choice Point に退避されている引数の数が 0 でなく，かつ命令キャッシュがヒットすれば，パイプラインは必ず再充填される。

なお，以上述べたようなパイプラインの挙動に関する予想は，実行するプログラムの性質や最適化手法に影響される。従って，実動作環境におけるパイプラインの挙動の評価や，それに基づく改良の検討は今後の課題である。

3.3.4 タグ・アーキテクチャ

PSI-III のタグ・アーキテクチャの中で，従来機と最も異なる部分は，パイプライン化されたタグ判定とデレファレンスである。この特徴を明確化するために，まず E ステージのタグ操作のための機構について述べ，その後これと対比する形でパイプライン化された機構とその効果について論ずる。

3.3.4.1 E ステージのタグ・アーキテクチャ

E ステージのデータ系に関しては，主要なレジスタに対するタグの付加，ALU 演算と並行して行うタグ比較，タグ即値生成とそれに対するマイクロ命令構成上の配慮など，PSI-II とほぼ同様のアーキテクチャとなっている。但し，DST-Bus に対するタグ即値出力だけではなく，S1-Bus についてもタグ即値出力機構を設け，レジスタ間の転送の並列化がタグ操作のために損なわれることがないように配慮した。また，KL1 処理における MRB の操作のために，ALU 演算と並行して MRB をオン／オフする機能を付け加えた。

順序制御系に関しては，3.3.1 で述べたように，タグと即値の比較結果による条件分岐が 2 フェーズに分割されたのが大きな変更点である。これと同様に，タグによる多方向分岐も 2 フェーズ化された。即ち，第一のフェーズでは `case_tag_setup` 操作により，MDR1/MDR2 のタグにより多方向分岐テーブル TGT が索引され，その結果がレジスタ TDAR に設定される。その後第二のフェーズで，TDAR の内容とベース・アドレスが OR されて分岐 `case_branch` が行われる。

この他に，デレファレンスのための三方向分岐が新たに設けられた。これは，AS1-Bus のタグ部が `ref` か，マイクロ命令で指定された即値に等しいか，またはそれ以外かによって 2 ビットのコードを生成し，それを TDAR のビット 3 ～ 2 に設定する機能である。この機能は，組込述語など複数のオペランドを持つものに関しては，E ステージでもデレファレンスを行う必要があるために設けられた。また KL1 の処理のために，`ref` である場合には MRB のオン／オフも判別する，四方向分岐の機能も備えられている。

3.3.4.2 タグ判定のパイプライン化

前述のように，E ステージにおけるタグ判定機構の 2 フェーズ化は，ユニフィケーションなどの処理にかなりのダメージを与える。例えば，`get_list` の処理は，オペランドが MDR1 にセットアップされているとし，かつそれがリスト・セルへのポインタであっても；

```
MDR1 = A[i] ;
_____________________________________________________
case_tag_setup(MDR1.tag) ;
case_branch {
        case list :   S = MDR1 , MODE_A = read , eop() ;
        case ref :    ...
        case undef:   ...
        ⋮
}
```

のように，3ステップの処理となる（横線の上の処理はパイプラインで行われることとする。以下同様）。また，list を優先して；

```
MDR1 = A[i] ;
_____________________________________________________
S = MDR1 , COND = (MDR1.tag==list) , MODE_A = read ;
case_tag_setup(MDR1.tag) , if (COND) eop() ;
case_branch {
        case ref :    ...
        case undef:   ...
        ⋮
}
```

とすれば2ステップの処理となるが，ref や undef の場合には先の例よりも1ステップ増加してしまう。

そこで PSI-III では，以下に示す三種類のタグ判定機構を S ステージに設けた。

(1) 任意のオペランドのタグと即値の比較結果の **COND** への設定
(2) メモリ・オペランドのタグによる多方向分岐セットアップ
(3) メモリ・オペランドのタグと即値の比較結果による実行開始アドレスの変更

まず，(1) に関しては比較的単純な機構であり，S ステージがなくても実現することができる。この機能を用いると，前述の get_list は；

```
MDR1 = A[i] , COND = (A[i].tag==list) ;
_____________________________________________________
S = MDR1 , case_tag_setup(MDR1.tag) , MODE_A = read ,
        if (COND) eop() ;
case_branch {
        case ref :    ...
        case undef:   ...
        ⋮
}
```

となり，1ステップ短縮される。なお，一段のデレファレンスのためにパイプラインでのメモリ・フェッチを行うことも考えられる。この場合，S ステージがなければパイプラインでの処理は；


```
addr = A[i] ;
if (addr.tag==ref) MDR1 = *addr ,
        else { MDR1 = addr , COND = (A[i].tag==list) ; }
_____________________________________________________
...
```

のようになり，フェッチしたデータが list であることは判定できないため，ref → list の場合は3ステップの処理となる。一方 S ステージがあれば；

```
addr = A[i] ;
if (addr.tag==ref) data = *addr , else data = addr ;
MDR1 = data , COND = (A[i].tag==list) ;
_____________________________________________________
...
```

となり，ref → list の場合も1ステップで処理できる。

次に，(2) のタグ多方向分岐のセットアップ機能は，list でない時の処理を高速化する効果がある。例えば；

```
addr = A[i] ;
if (addr.tag==ref) data = *addr , else data = addr ;
MDR1 = data , case_tag_setup(data.tag) ;
_____________________________________________________
case_branch {
        case list :   S = MDR1 , MODE_A = read , eop() ;
        case ref :    ...
        case undef:   ...
        ⋮
}
```

とすれば，list である時の処理は2ステップとなるが，ref や undef の場合の処理は更に1ステップ短縮される。

これに (3) のマイクロプログラム実行開始アドレスの変更機能を組み合わせると，list である時の処理を1ステップで実現できる。即ち；

```
addr = A[i] ;
if (addr.tag==ref) data = *addr , else data = addr ;
MDR1 = data , case_tag_setup(data.tag) ,
        if (data.tag==list) start(for_list) ;
        else start(for_others) ;
_____________________________________________________
for_list :
        S = MDR1 , MODE_A = read , eop() ;
for_others :
        ...
```

表 3-6: get_list のステップ数

case	PSI-Ⅲ	PSI-Ⅱ
list	1	1
ref → list	1	4
ref → undef (no trail)	2	5
ref → undef (trail)	3	7

とすることができる。

なお，(1) と (2) の組合せをマイクロ操作；

```
if (COND) eop() , else case_branch ;
```

により実現し，list の場合を 1 ステップとすることも考えられるが，ハードウェアが複雑化することや一般性がないことなどから得策ではないと判断した。

この結果，get_list の処理は引数が list である時はもちろん，それ以外のケースについても高速に行われる。実際，表 3-6に示すように PSI-Ⅱ と比較すると，list である場合は共に 1 ステップ，その他の支配的ケースである ref → list/undef* については，それぞれ 3 ステップ（undef でトレイルが必要な場合は更に 1 ステップ）短縮されている。

なお (1) と (3) の組合せも可能であり，例えば加算を行う組込述語 “add Ai,Aj,Ak” は；

```
addr = A[i] ;
if (addr.tag==ref) data = *addr , else data = addr ;
MDR1 = data , MDR2 = A[j] , COND = (A[j].tag==int) ,
       if (data.tag==int) start(for_int) ,
       else start(for_others) ;
-------------------------------------------------------
for_int :
       XR = MDR1 + MDR2 ,
            if (! COND) { "dereference or exception" ; }
       A[k] = XR , if (! overflow) eop() ;
        ⋮
```

のように，A_j が整数であれば 2 ステップで完了する。

*正しくはグローバル・スタック上の undef。

この他，S ステージのタグ操作機能として，タグ即値を付加したオペランド・セットアップがある。例えば unify_variable の処理は，この機能と MODE_A による実行開始番地の変更，及び G/S のセットアップ機能を用いて；

```
MDR1.tag = undef ,
       if (MODE_A==read) { MDR1.value = S , start(for_read) , }
       else { MAR1.value = MDR1.value = G , start(for_write) ; }
-------------------------------------------------------
for_read:
       A[i].tag = ref ; A[i].value = MDR1 ; eop() ;
for_write:
       *MAR1 = MDR1 ; eop() ;
```

のように，読出／書込モードとも 1 ステップで実現されている。なお，MODE_A による実行開始アドレスの修飾機能を有効に利用するためには，S ステージのもう一つのタグ操作機能である，タグ判定結果による MODE_A の設定機能が必要である。即ち，実際の get_list の S ステージでの処理は；

```
MDR1 = data , case_tag_setup(data.tag) ,
       if (data.tag==list) { MODE_A = read , start(for_list) , }
       else { MODE_A = write , start(for_others) ; }
```

となっており，この直後に出現する *unify* 系命令が MODE_A を参照できるようにしている。

3.3.4.3 デレファレンスのパイプライン化

前述のように，S ステージを設けたことにより，パイプライン化された強力なタイプ判定機構を実現することができた。S ステージにはこの他に，メモリ・オペランドのタグが ref ではなくなるまで，その指示するデータを繰返しフェッチするデレファレンス機能がある。この機能は，RF をルートとする場合には連鎖が 2 以上の時に，また局所変数などメモリ上のオペランドをルートとする場合には連鎖が 1 以上の時に，それぞれ作動する。

デレファレンスのためのハードウェア機構，即ちメモリ・オペランドのタグが ref であることの判定と，その値によるメモリ読出を行う機構は比較的単純であり，かつ以下のようなメリットがある。

(1) ハードウェア化により 1 ポインタあたり 2 サイクルの高速デレファレンスが可能となる。

(2) E ステージとの並行動作による性能向上が期待できる。

(3) オペランドが完全にデレファレンスされることにより，マイクロプログラムがコンパクトになり，かつ書き易くなる。

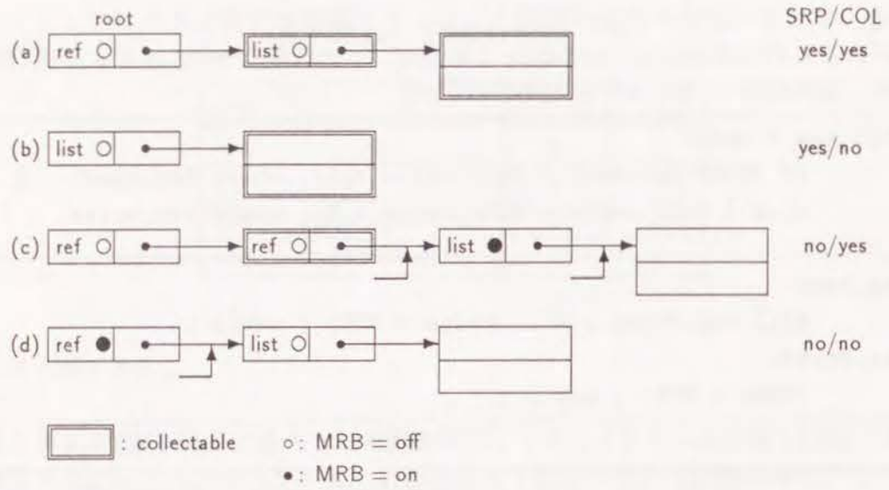


図 3-23: MRB によるガベージ・コレクションのサポート

更に，付録 B に示す MRB による実時間ガベージ・コレクションのサポートをデレファレンス機構に組込むことにより，KL1 のデレファレンス処理の効率が大幅に改善されている。即ち S ステージは，デレファレンスの結果として；

SRP　連鎖中の MRB が全てオフ
COL　最初の二つのポインタの MRB がオフ

の二つの情報を E ステージに引渡す。これらの組合せにより E ステージは，図 3-23に示すように，Reference Pointer の連鎖や終端のデータの回収の可否を知ることができる。

なお，デレファレンスは一種の間接アドレスによるオペランド・フェッチであるため，アドレッシングに用いたメモリ上のデータが書換えられることを考慮しておかなければならない。しかしこの問題は，論理型言語の持つ「単一代入」の性質を用いて，簡単に解決される。即ち，KL0/KL1 の処理では，バッチ的なガベージ・コレクションを除くと，Reference Pointer が書換えられるのは以下の場合に限られる。

(a) KL0 のバックトラック時に Reference Pointer が未定義変数に Undo される。
(b) KL1 の実時間ガベージ・コレクションにより Reference Pointer が回収される。

さて (a) については，バックトラック時に最初に実行される命令である `retry(_me_else)` や `trust(_me)` がデレファレンスを行わないため，問題が生じない。即ち，デレファレンスを行う命令は，Undo が完了するまで S ステージには到達せず，かつ D ステージでのオペランド・フェッチも抑止されるため，Undo される前の Reference Pointer が読出されることはない*。

一方 (b) については，Reference Pointer の回収を連鎖の先頭部分に限ることによって解決する。即ち，連鎖の先頭部分に存在する回収可能な Reference Pointer に対して，他の参照パスが存在しえないことは，MRB の定義から明らかである。なお，回収の対象をポインタ連鎖の先頭部分のみに限定しても，メモリ使用効率はほとんど変わらないことが知られている [Kimura 90]。

3.3.4.4　パイプライン化の効果

以上述べたように，PSI-III ではタグ判定とデレファレンスをパイプライン化するために，通常のパイプライン・プロセッサにはないステージである S ステージを設けた。この方式がどの程度有効であるかを調べるために，3.3.1 で述べた様々な選択肢に関して，*append* の性能をシミュレーションにより求めた。選択肢として選んだアーキテクチャは，以下のものである。

(1) **PSI-II**
PSI-II ではタグ判定／デレファレンスは以下の操作により行われる。
(a) レジスタ・ファイルの読出
(b) 読出したデータのタグと即値の比較
(c) 比較結果に基くマイクロプログラムの分岐先アドレス生成
(d) マイクロ命令のフェッチ
この操作は 1 マシン・サイクル (155ns) で行われるが，この操作のためのパスがクリティカル・パスとなっている。

(2) **Device Speed Up**
PSI-II の論理素子を，PSI-III と同等の速度のとしたもの。マシン・サイクルは 100ns 程度と想定される。

(3) **Two Phase Execution**
(1) のクリティカル・パスを (a) と (b)，及び (c) と (d) の 2 フェーズに分割したもの。この構成は E ステージと同様であり，60ns のマシン・サイクルで動作することができる。

(4) **Two Stage Pipeline**
(a) と (b) の操作のために機械命令レベルのパイプライン・ステージを設けたもの。

*バックトラック直後の命令がデレファレンスをするような処理方式にも耐えうるようにするために，デレファレンスのインタロック機構も備えられている。

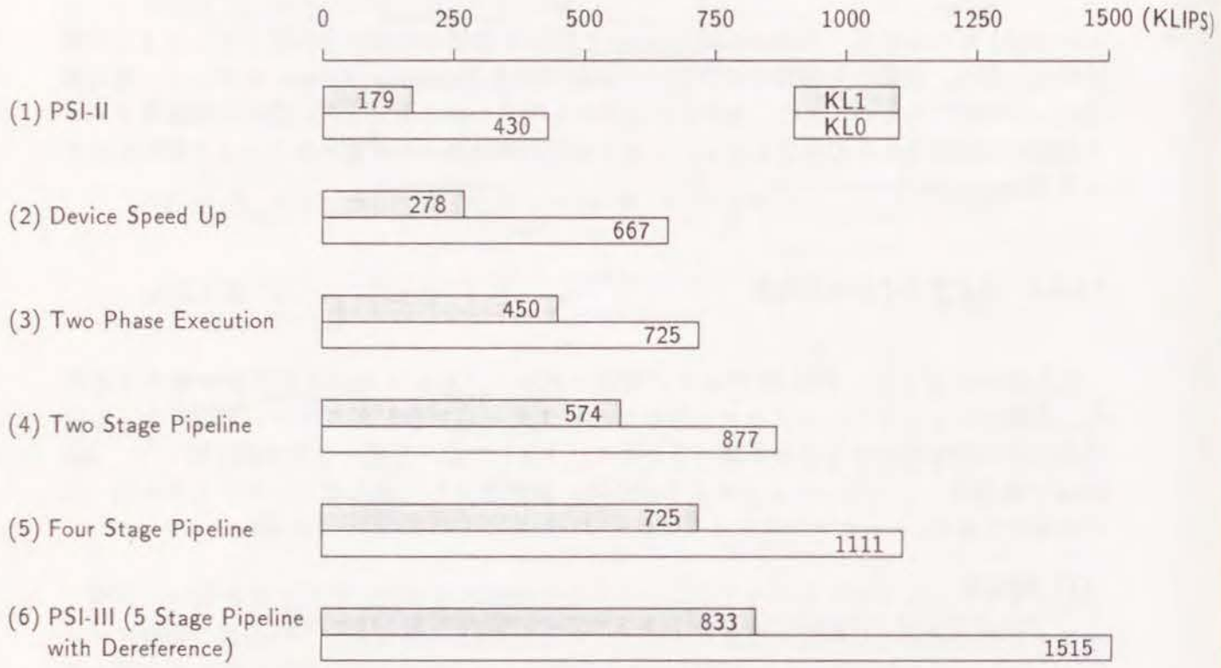


図 3-24: パイプライン・アーキテクチャによる性能向上

この構成はS及びEステージを組み合わせたものとほぼ同様であるが，デレファレンスはパイプライン化されていない。

(5) **Four Stage Pipeline**
(4) にアドレス計算とオペランド・フェッチのステージを付加したもの。一段のデレファレンスは行われるが，その結果に対するタグ判定はEステージで行われる。

(6) **PSI-III**
デレファレンスがパイプライン化されている。また，デレファレンス結果に対するデータ・タイプ判定の場合，Eステージでの条件分岐が不要となるので，(4) や (5) よりも強力である。

これらの各々について，KL0/KL1 の *append* の性能をシミュレーションにより求めた値を図 3-24に示す。この評価結果から，パイプライン化されたタグ判定／デレファレンスが，マシン・サイクルとサイクル数の両面で性能向上に大きく寄与していることが明らかである。

3.3.5 メモリ・アーキテクチャ

3.3.5.1 アドレス変換バッファ (TLB)

メモリ・アーキテクチャに関するPSI-IIからの最大の変更点は，従来完全にハードウェア化されていたアドレス変換機構を，TLBに置き換えたことである。この変更は，ハードウェア量の削減のために行ったものであるが，これによる性能の低下を最小限に留めるために，様々な工夫を行っている。

まずプロセスが異なれば，同じ論理アドレスが異なる意味を持つ多重論理空間は，TLBの各エントリに3ビットのプロセスID (*pid*) を付加することにより実現した。即ち，マイクロプログラムは最近実行した8個のプロセスの番号とpidのマップを保持し，プロセス切換時にこれを検索してハードウェアに対して実行中のプロセスのpidを与える。従って，実行すべきプロセスがマップ中に存在しない場合には，LRUによってマップを変更するとともに，TLBから古いpidを持つエントリを追い出す必要がある。しかし，PSI-IIを用いて評価したところ，マップのヒット率は99.99%以上であり，マップ・ミスによるオーバヘッドは1%以下であることが明らかになっている。

次に，TLBの構成は命令／データ用とも2セットのセット・アソシアティブ方式とし，容量はそれぞれ64エントリとした。またデータ用TLBについては，各スタック及びヒープのアクセス特性を考慮して，そのエントリ・アドレス $EA\langle 4:0\rangle$ を；

$$EA\langle 4:2\rangle = page\#\langle 4:2\rangle \oplus area\#$$

$$EA\langle 1:0\rangle = \begin{cases} page\#\langle 1:0\rangle, & \text{for heap} \\ page\#\langle 1:0\rangle \oplus pid\langle 1:0\rangle, & \text{for stacks} \end{cases}$$

とした（但し'⊕'はExclusive-OR）。これは，一つのTLBエントリを；

- 一つのプロセスの複数のスタック
- 複数のプロセスの同じスタック

が奪い合うことを防止する効果がある。これらの結果，5.3.3で示すように，TLBのヒット率を実質的に100%とすることができた。

なおTLBミスの処理は，マイクロプログラムにより行うこととし，このためにマイクロプログラム・レベルでのトラップ機構を導入した。即ちTLBミスが発生すると，その直後のマイクロ命令アドレスが復帰アドレスとして自動的にマイクロプログラム・スタックにプッシュされ，トラップ・ルーチンへの分岐が行われる。また，ミスの原因となった論理アドレス，読出／書込の区別，書込データ，読出のデスティネーションが**MDR1/2**のどちらであったか，など必要な情報を全て自動的に退避する機構が備えられており，トラップ・ルーチンでのTLBの更新や再アクセスが行えるようになっている。

3.3.5.2 ハーバード・キャッシュ

パイプライン化に伴うメモリ・アクセス頻度の増加に配慮した，命令キャッシュとデータ・キャッシュの分離，即ちハーバード・アーキテクチャの採用も，PSI-Ⅲの大きな特徴の一つとなっている。この方式を実現するために必要なハードウェアはかなりの量であり，その効果がどの程度のものであるかには重大な関心が持たれる。そこで，二つの方法によって分離の効果，即ち命令とデータの並行アクセスによる効果を見積もった。

まず最初に，一命令に要する平均サイクル数を3，またデータの読み書きのためのメモリ・アクセス頻度を50%として，命令とデータ・アクセスの競合の頻度を見積もった。この場合，並行アクセスが可能であれば6サイクルの間に命令フェッチが2回，データ・アクセスが3回発生する。一方，競合の確率は；

- 0回 $= 1/5$
- 1回 $= 3/5$
- 2回 $= 1/5$

と計算することができ，競合のための損失は；

$$(1 \times 3/5) + (2 \times 1/5) = 1$$

となる。従って，並行アクセスによる効果は $1/6 = 16.7\%$ であると見積もられる。

次に，*append*の再帰呼出クローズの実行過程を，並行アクセスの可否それぞれについてシミュレーションした。このクローズの一回の実行には，命令フェッチ及びデータ・アクセスがそれぞれ7回ずつ行われるが，並行アクセスが可能な場合には11サイクルで完了する。一方，並行アクセスができない場合は最低14サイクルは必要となり，効果は $3/11 = 27.3\%$ と見積もられる。実際には，並行アクセスができないと命令の供給が実行に追いつかなくなるため，更に1サイクルの損失が発生し，効果は $4/11 = 36.4\%$ に拡大する。

以上を総合すると，並行アクセスの効果は20%程度はあるものと見積もられ，ハードウェアを投資した価値は充分にあったものと考えられる。

3.3.6 マイクロ命令アーキテクチャ

PSI-Ⅲでは，**E**ステージが2ステージ化されたために，並行動作が可能なハードウェア機構がPSI-Ⅱよりもかなり増加した。これらのハードウェアを有効に利用するためには，マイクロ命令のビット数を増やして並行制御を可能にしなければならない。実際，PSI-Ⅲのマイクロ命令は図3-25に示すように，PSI-Ⅱの53ビットから11ビット増加し，64ビット幅となっている。しかし，個々のフィールドに関してはPSI-Ⅱで導入したエミット・フィールドの考え方を更に押し進めてビット幅を圧縮し，全体の増加量を最小限のものとした。

(1) DBGF

PSI-Ⅱと同様，ブレーク・ポイントの設定と評価用カウンタ**GEVC**のインクリメントを行う。

(2) CCF

メモリ・アクセス操作と，アドレス／データ・レジスタの選択，及びアドレス・レジスタのインクリメント制御を行う。PSI-Ⅱに比べてアドレス／データ・レジスタの可能な組合せを増やして使いやすくしたことと，4.2.2で述べるトレイル・バッファの操作などのために命令コードが増加し，フィールドの幅は1ビット増えて5ビットとなった。

(3) WRBF/DSTF/SC1F/SC2F

フェーズ1のレジスタの書込(**DSTF**)と，S1/S2-Busへの読出(**SC1F**/**SC2F**)を制御する。各フィールドのビット数は，以下の工夫によってPSI-Ⅱから1～2ビット削減されている。

まず，コードの大半を占める**WR**のアドレス（5ビット）の内の1ビットを，各フィールド共通に**WRBF**が定めることとした。但し，共通化できない場合のために，**DSTF**/**SC1F**では**EM1**が，**SC2F**では**EM2**をアドレスとするコードを設けた。

1	5	1	5	5	5	3	2	3	3	2	2	4	4	1	2	6	6	4
DBGF	CCF	WRBF	DSTF	SC1F	SC2F	SD1F	SD2F	ADF	AS1F	AS1F	ALF	JMPF	CNDF	MCF	TGIF	EM1	EM2	EM3

図 3-25: PSI-Ⅲのマイクロ命令

また，**SC2F** では命令レジスタ **IR** のフィールド抽出が指定できるが，具体的なビット数や位置の指定は **EM2** によるものとした。更に，順序制御やパイプライン制御のための雑多なレジスタ類に関する読出／書込については，**EM1** によってレジスタを指定することとし，**DSTF/SC1F** のコードを削減した。

(4) SD1F/SD2F

S1/S2-Bus を経由したフェーズ1からフェーズ2への転送のデスティネーションを選択するフィールドであり，2フェーズ化のために新設された。なお，同時書込操作の有効性を考慮して，**SD1F** のコードには **MAR1** と **MDR1**，及び **MAR2** と **MDR2** へ，同じ値を設定する操作が含まれている。

(5) ADF/AS1F/AS2F

DST-Bus からフェーズ2のレジスタへの書込，及び AS1/AS2-Bus への読出を制御するフィールドであり，2フェーズ化のために新設された。**AS1F** と **AS2F** については，原則的には AS1/AS2-Bus をそれぞれ制御するものであるが，定数生成など特殊な操作については **AS1F** が **AS2F** の意味を修飾するようにして，最低限のビット数で意味のある組合せがすべて実現できるようにした。また **ADF** については，**MAR1** と **MDR1** への同時書込操作を盛り込み，**DSTF** と併せて同時に3つのレジスタに同じ値を書込むことができるようにした。

(6) ALF

ALU で行う演算の種類を定めるフィールドである。新設された演算であるフィールド抽出機能や MRB のオン／オフなど，加減算以外の演算は全て **EM2** が定めることとし，ビット数の増加を防いだ。

(7) JMPF/CNDF

分岐タイプの指定 (**JMPF**) と，分岐条件の指定 (**CNDF**) を行うフィールドである。分岐条件生成の一部を **CNDF** から **JMPF** に移すとともに，使用頻度が比較的少ない条件の選択は **EM1** が行うこととして，合計のビット数を2ビット削減した。

(8) MCF/TGIF

PSI-II では **TYPF** が担当していたカウンタ／フラグの制御とタグ即値生成指定を，前者は **MCF** に，後者は **TGIF** に分離した。更に DST-bus に対してはタグ即値に **EM2** も使用できるようにして，並行制御の範囲を拡大した。

3.4 並列推論マシン

前節までに述べた3つの逐次型推論マシンは，並列推論マシンの要素プロセッサとしても用いられている。最初に開発した並列推論マシンである Multi-PSI/v1 [Masuda 88] は，6台の PSI-I を簡単なハードウェアを用いて接続したものであり，KL1 の処理もソフトウェアで行うなど，小規模かつ実験的なものであった。但し，プロセッサ間でのユニフィケーションなど，基本的な並列処理の方式を検討し，かつ実動作環境で検証できたことは，以後の本格的な並列推論マシンの研究に多いに役立った。

次に開発した Multi-PSI/v2 [Takeda 88, Uchida 88] は，PSI-II の CPU を要素プロセッサとして最大64個結合した，かなり大規模な並列推論マシンである。処理速度の面でも，KL1 の処理がマイクロプログラム化されたことや，プロセッサ間通信のためのハードウェアの充実により，Multi-PSI/v1 を遥かに凌駕するものとなっている。これに加え，オペレーティング・システム PIMOS [Chikayama 88] が実装されたことにより，並列応用プログラムの開発環境が整い，広汎な応用分野における並列ソフトウェアの研究が可能となった [ICOT 90]。

更に，現在開発中の PIM/m [Nakashima 90c, 90d] では，PSI-III の CPU を要素プロセッサとすることにより，プロセッサ数を最大256に拡張した。また，知識ベースなどの大量のデータを扱う応用分野へも並列プログラミングを適用できるように，大容量のディスク・システムを新たに装備した。

以下，Multi-PSI/v2 と PIM/m に関して，そのシステム構成とプロセッサ間通信の機構を論ずる。

3.4.1 システム構成

Multi-PSI/v2 と PIM/m のシステム構成を，図 3-26と図 3-27にそれぞれ示す。いずれのシステムにおいても，要素プロセッサは二次元のメッシュ状のネットワークにより接続されており，プロセッサ間のユニフィケーションなどはメッセージ通信により行う疎結合システムとなっている。また，各要素プロセッサは 16 Mw という大容量の「ローカルな」主記憶を持っており，大規模な並列プログラムの実行にも充分に耐えうる構成となっている。

ユーザとのインタフェースなどの入出力処理は，フロント・エンド・プロセッサ (FEP) である PSI-II/PSI-III を介して行われる。FEP と本体間の通信は，Multi-PSI/v2 ではプロセッサ間ネットワークと同じものが，PIM/m では SCSI が用いられる。また PIM/m では 8 プロセッサに一つの割合で，600 MB のディスクが SCSI バスを経由して接続されており，大規模知識ベースの構築などが可能となっている。

実装に関しては，Multi-PSI/v2 では 8 プロセッサ，PIM/m では 32 プロセッサを格納した筐体を単位として増減が可能であり，小／中規模な構成から最大構成まで，必要に応じて様々なシステムを構築することができる。また，FEP の数も 1 ～ 4 と可変であり，大規模な構成では FEP についても負荷分散を行うことができる。更に，FEP に接続された LAN を経由した遠隔利用も可能で，多数のユーザが計算資源を共有できるように配慮されている。

なお，PSI-II や PSI-III は FEP として用いられるだけではなく，KL1 プログラムのデバッグ・ベンチとしても使用することができる。即ち，KL0 と KL1 の双方の処理系を実装した pseudo Multi-PSI や pseudo PIM といったシステムを構築することにより，手軽でかつ高速なデバッグ環境を提供している。またこれらのシステムでは，複数の要素プロセッサの動作をシミュレートする擬似並列実行も可能であり，並列化に伴う非決定性がもたらすバグを早期に検出することができる。

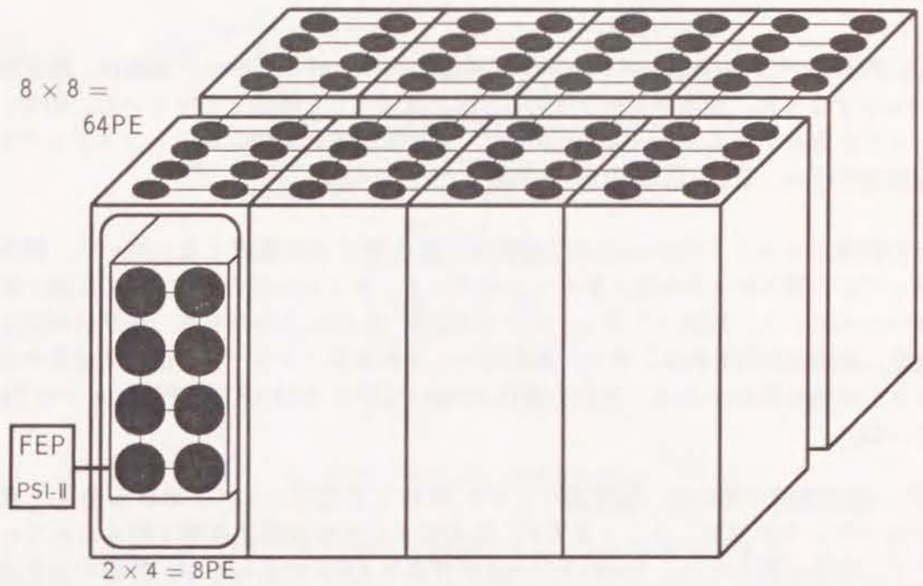


図 3-26: Multi-PSI/v2 のシステム構成

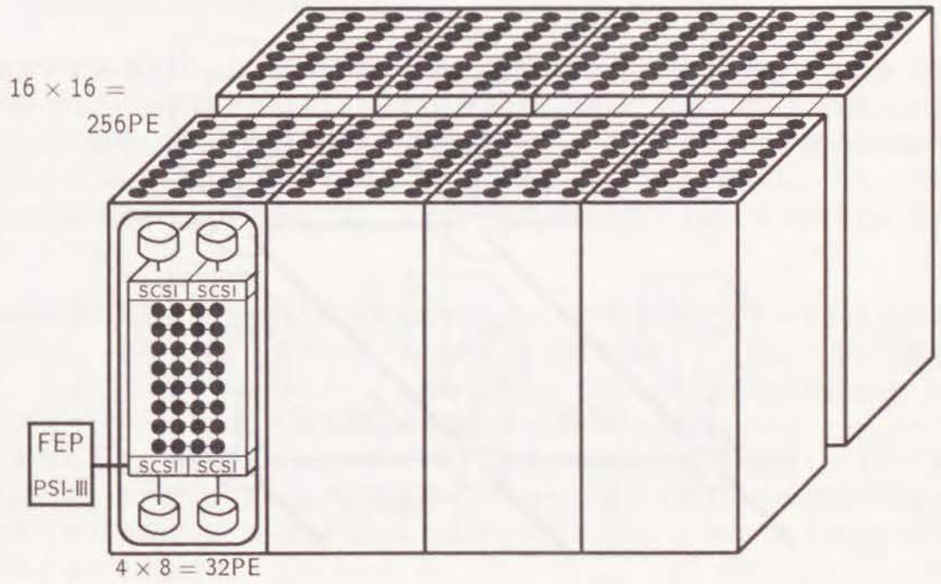


図 3-27: PIM/m のシステム構成

3.4.2 プロセッサ間通信機構

Multi-PSI/v2 や PIM/m のプロセッサ間通信のためのハードウェア機構は，概念的には図3-28に示すように，要素プロセッサの CPU とは独立した構造となっている。即ち，要素プロセッサを通過するメッセージについては，その転送方向の決定やバッファリングを通信機構が独自に行い，CPU の処理は一切不要となっている。

個々の要素プロセッサのための通信機構は，図3-29に示す構成となっている。隣接するプロセッサとの間のチャネルは，8ビットのデータ，メッセージ・パケットの先頭／末尾を示すマーク・ビット，及びパリティ・ビットの合計10ビットからなり，送受信が完全に独立した全二重通信が行われる。また，各送信ポートにはネットワークの渋滞を緩和するためのバッファが備えられており，その容量は Multi-PSI/v2 では 48 B，PIM/m では 64 B となっている。

CPU と通信機構の間には，送信用バッファ **WB** と受信用バッファ **RB** があり，**RB** についてはパケットが到着したことを CPU に通知するための割込機構も備えられている。バッファの容量に関しては，Multi-PSI/v2 では各々 4 KB であったが，評価の結果過大であることが明らかとなり [Nakajima 90a, 90b]，PIM/m では 1 KB に削減されている。その一方で，パケットの組立や分解の手間を削減が必要であることが指摘されたため，40 ビット・ワードと 8 ビット・バイトの相互変換のための機構が PIM/m では追加された。

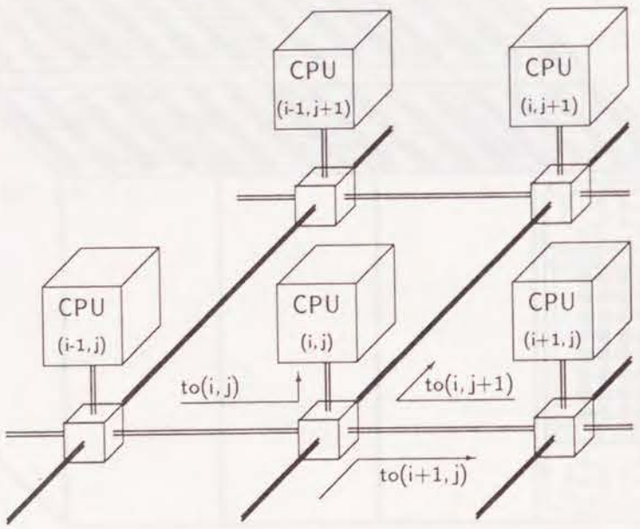


図 3-28: ネットワークの構造

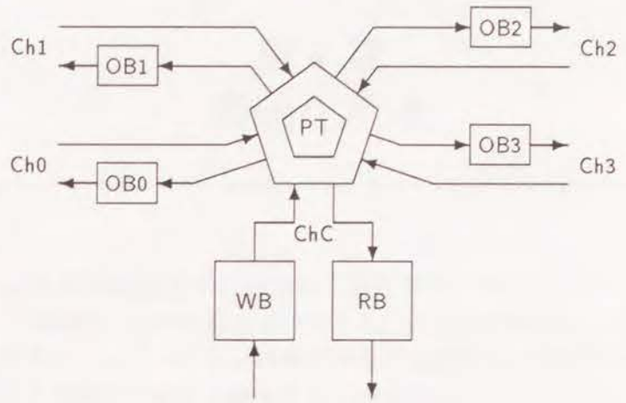


図 3-29: プロセッサ間通信機構

パケットのスイッチングは，転送方向を決定するためのテーブル **PT** と，5×5 のスイッチ回路を用いて行われる。即ち，パケットの先頭に記された宛先により **PT** を読出し，4つのプロセッサ間チャネルまたは **RB** のいずれに転送するかを決定する。スイッチ回路は一種のクロスバーであり，転送先が競合しない限り，最大五つの入力パケットを同時に転送することができる。また，**PT** も入力ポートの数に応じた5ポート読出が可能であり*，**PT** の読出競合による遅延も発生しない。

なお **PT** を RAM とし，更にそのエントリ数をプロセッサ数よりも大きくしたことにより，様々な応用が可能となっている。例えば Multi-PSI/v2 では，**PT** を 16 K エントリとして，動的負荷分散のための「論理プロセッサ・アドレス」による通信の試験的実装を可能とした [Takeda 88]。また PIM/m においても容量を 1 K エントリ確保し，パケットの種類によって転送経路を変更するなどの，プロセッサ間通信に関する様々な実験を行えるように配慮している。

通信機構の実装は，Multi-PSI/v2 では 20K ゲートの CMOS ゲート・アレイ2チップを中心に行い，各種バッファやテーブルのための SRAM を含めて，1枚のプリント基板に収めた。従って，CPU のためのプリント基板3枚と，4枚の主記憶ボードを合わせ，一つの要素プロセッサは8枚のプリント基板から構成される。一方 PIM/m では，バッファ／テーブルを含めた全ての回路を 1 μm の CMOS VLSI 上に搭載し，CPU と同じプリント基板上に実装した。その結果，主記憶ボード1枚と合わせて，要素プロセッサの実装規模はプリント基板2枚に圧縮され，Multi-PSI/v2 とほぼ同じ大きさのシステム内に4倍のプロセッサを収容することができた。

*Multi-PSI/v2 では CPU からの送信パケットには陽に転送路が指定されるため4ポート読出となっている。

第4章
最適化手法

第2章で述べたように論理型言語の処理は，述語呼出，ユニフィケーション，バックトラックといった，「高度な」操作に基くものである。これらの操作は「一般的」には複雑なものであり，例えばユニフィケーションは構造体を含む任意の二つのデータに対して，再帰的に比較や代入を行う操作であると定義することができる。

しかし，これらの操作を単純な分岐や代入などに置き換えることができることがあり，またそのようなケースは出現頻度はかなり高い。即ち，論理型言語と言えどもプログラミング言語である以上，従来の手続型言語で頻繁に行われるループやif-then-elseのような制御，また変数への代入や簡単な演算などのデータ操作は，やはり高い頻度で実行される。従って，ユニフィケーションなど「一般的」には複雑である操作に潜む，簡単でかつ出現頻度の高い部分を見い出し，それを高速に実行することが論理型言語処理の高速化のポイントであると言うことができる。またこれを更に進めて，出現頻度が低い複雑な操作の実現のために，簡単で高頻度の操作の性能を低下させてはならない，という議論も充分に成立する。

実際，2.2で述べたユニフィケーションの実現手法や，2.4で述べた基本的な最適化手法であるTRO, FGO, Clause Indexingは，全てこの「簡単で高頻度の操作を高速に」という発想に基くものである。この発想が正しいものであることは，これらの手法がPSI系列の推論マシンを含む多くの処理系で採用され，かつ高い性能を達成するために大きく貢献していることにより証明されている。

しかし，実用的なプログラムを真に高速化するには，これらの基本的手法のみでは必ずしも充分ではない。例えば，一般の応用プログラムでは，述語呼出の半数以上が組込述語の実行に占められており，その実現手法の良否が総合的な性能を大きく左右することは明らかである。また，Clause Indexingを用いても「決定的」にクローズを選択することができないケースも数多くあるが，これらに対しても最適化の可能性は充分に存在する。更に，システムとしての実用性を高めるために導入した，KL0の実行順序制御機能などがもたらす「例外的事象」を，「通常」の処理の速度を低下させることなく実現することも重要な課題である。

これらの項目，即ち組込述語，バックトラック，及び例外処理に関して，筆者らは様々な最適化手法を考案し，PSI-II及びPSI-IIIに実装してその効果を実証した。以下本章では，

各項目に関して処理方式の詳細を述べるとともに，「簡単で高頻度な操作を高速に」という発想がどのように生かされているかを論ずる。なお，最適化の効果に関しては本章では定性的な議論に留め，定量的なものに関しては5.2節において論ずることとする。また，本章に関する研究は，近山隆，近藤誠一，立野裕和らの協力を得て行った。

4.1 組込述語の最適化

4.1.1 組込述語の性質

KL0にはデータ・タイプ判定，構造データ操作，算術／論理演算及び比較など，150種以上の組込述語が備えられている。これらは実用的なプログラムの作成に不可欠のものであるばかりではなく，極めて頻繁に使用されることが知られている。例えば，前述の *Window* や *BUP* では，述語呼出に占める組込述語の割合がそれぞれ82%と65%と極めて高く[Nakajima 86a, Nakashima 87b]，その他の様々なプログラムにおいても半数以上を占めることが明らかになっている[Tateno 89]。従って，実用的なプログラムの実行性能を高めるためには，組込述語に関する処理の高速化が極めて重要であることは明らかである。

さて，多くの組込述語の機能は比較的簡単なものであり，最適化対象の基本条件である「簡単で高頻度」を満たすものである。例えば加算を行う組込述語 `add(X,Y,Z)` の機能は，基本的には `X` と `Y` を加えることであり，引数のデレファレンスやタイプ・チェックを含めても，さほど複雑な操作は必要としない。従って，`add` の機能をマイクロプログラムで実現することは比較的容易であり，また処理速度の面でも得策であるといえる。実際，PSI系列の推論マシンではほとんど全ての組込述語がマイクロプログラムで処理されている。特にPSI-IIやPSI-IIIでは，呼出しの手間は最小化するために，組込述語を機械命令として表現している。この「マイクロプログラム化」と「命令化」が，組込述語の最適化の第一ステップである。

また，組込述語の処理が「簡単」であるということは，コンパイラがその挙動を（ほぼ）完全に予測できるという意味も持っている。即ち，組込述語の処理の中では；

- 他の述語を呼出さない
- Environment や Choice Point を生成しない

ため，グローバル・スタック・トップ **G** とトレイル・スタック・トップ **TR** 以外のレジスタは，一切変更されることがない。この性質は，例えば `get_list` のようなヘッド・ユニフィケーション命令と同じである。従って，':-' の直後に出現する組込述語は，ヘッド・ユニフィケーションの一部であるとみなすことができる。

この性質を用いて，2.4で述べたFGOを拡張することができる。FGOでは；

```
p(X):- q(X,Y), r(Y).
```

における変数 `X` は，ヘッドと第一ゴールにのみ出現するため，一時変数として引数レジスタに割付けることができる。この割付規則を拡張して；

> ヘッド，':-' の直後の組込述語，及び最初の「通常」ゴールにのみ出現する変数は，引数レジスタに割付ける。

とすることができる。例えば；

```
p(X,Y,Z):- W is X + Y, q(W,Z,U), r(U).
```

では，X, Y, Z, W を引数レジスタに割付けることができる。また，組込述語が複数ある場合も同様であり；

```
p(X,Y,Z):- W is X + Y, W < Z, q(W,Z,U), r(U).
```

の変数割付も同じように行うことができる*。

この変数割付の最適化は，クローズの先頭部分以外でも実施することができる。例えば；

```
p(X):- q(X,Y,Z), W is Y + Z, r(W).
```

における W は，引数レジスタに割付けてもなんら問題がない。従って，変数割付規則をさらに拡張し；

> 組込述語（の列）とその直後の通常ゴールにのみ出現する変数は，引数レジスタに割付ける。

とすることができる。

また，FGO の拡張は変数割付だけではなく，TRO と組み合わせた Transitive Clause や Unit Clause の最適化にも適用できる。即ち；

> ゴールの数が高々一個であるクローズでは Environment を生成しない。

という規則を拡張し；

> ':-' の直後に出現する組込述語（の列）を除いたゴールの数が高々一個であるクローズでは Environment を生成しない。

とすることができる。この結果，例えば *Quick Sort* の一部である；

```
partition([X|L1],Y,[X|L2],L3):- X < Y, !, partition(L1,Y,L2,L3).
```

は，Transitive Clause とみなされる。また；

```
avarage(X,Y,A):- A is (X + Y)/2.
```

は Unit Clause とみなされる。

なお，通常ゴールの後に組込述語がある場合には，通常ゴールが一つしかなくても Transitive Clause とはみなされない。例えば；

---

*'W is X + Y' や 'W < Z' は一種のマクロであり，それぞれ add(X,Y,W), less_than(W,Z) に変換される。


```
p(X,Z):- q(X,Y), Z is Y + 1.
```

は q の実行完了後もクローズの処理を継続しなければならないため，Environment の生成が必要となる。但し，一般の複数ゴールを持つクローズが；

allocate + call + call + ... + call + deallocate + execute

とコンパイルされるのに対し，組込述語が最後のゴールとなっているクローズは；

allocate + call + call + ... + call + *built-in* + deallocate + proceed

とコンパイルされる。

4.1.2 引数の受渡し

組込述語の最適化の重要なポイントの一つに，引数の受渡しの方法がある。まず問題となるのは，add(X,Y,Z) の Z に関しては，加算結果とのユニフィケーションが必要なことである。従って，通常のゴールと同様に *put* 系命令によって引数を渡すと，add の処理の中でユニフィケーションが必要となる。また，Z が初出の未定義変数である場合*，変数を置いておく「場所」が必要となる。この「場所」をローカル・スタックに割当てると，Transitive/Unit Clause の概念を拡張して Environment 生成を省略する，FGO と TRO を組み合わせた最適化の適用範囲が狭められる。また，グローバル・スタックに割当てることもできるが，バックトラックでしか縮退できないグローバル・スタックを，無闇に伸ばすことは望ましくない。

そこで，組込述語の引数を入力型と出力型に分けて取扱う方法を考案した [Nakashima 86, 87c]。まず，add の X と Y のように，その値が組込述語の処理に必要な入力型の引数は，通常のゴールと同様に *put* 系命令で引数レジスタに値を設定する。一方，Z のように組込述語が処理結果を返す出力型の引数に関しては，*get* 系命令を用いることとした。即ち，組込述語が処理結果を単に引数レジスタに設定した後，引数の種類に応じた *get* 系命令によってユニフィケーションを行う。従って；

```
(1) p(X,Y):- Z is X + Y, ...
(2) p(X,Y,Z):- Z is X + Y, ...
(3) p(X):- Y is X + 1, ...
(4) p(X,Y):- Y is X + 1, ...
```

のための add は，それぞれ；

(1) put_value(*for* X) + put_value(*for* Y) + add + get_variable(*for* Z)
(2) put_value(*for* X) + put_value(*for* Y) + add + get_value(*for* Z)
(3) put_value(*for* X) + put_integer(*for* 1) + add + get_variable(*for* Z)
(4) put_value(*for* X) + put_integer(*for* 1) + add + get_value(*for* Z)

とコンパイルされる。

なお，ゴールとして出現するユニファイア 'X=Y' に対するコードは，この引数受渡し方式を応用して生成される。即ち，各々の引数が定数，構造体，変数のいずれであるか，また変数である場合には初出／既出，局所／一時のいずれであるかに応じて，表 4-1に示すような最適化されたコード生成が行われる。

さてもう一つの問題は，引数の受渡しをどの引数レジスタを用いて行うかである。単純な

*これが支配的である。

表 4-1: ユニフィケーション・ゴールのコード生成

X \ Y	定数	構造体	初出局所変数	初出一時変数	既出局所変数	既出一時変数
定数	fail or *none*	fail	put_const + get_var	put_const	put_val + get_const	get_const
構造体	fail	put_vect + get_vect	put_vect + get_var	put_vect	put_val + get_vect	get_vect
初出局所変数	put_const + get_var	put_vect + get_var	*1	*1	*1	*1
初出一時変数	put_const	put_vect	*1	*1	*1	*1
既出局所変数	put_val + get_const	put_val + get_vect	*1	*1	get_val	get_val
既出一時変数	get_const	get_const	*1	*1	get_val	get_val

*1 同じ変数とみなされる。
*2 リストについては構造体の get/put_vector を get/put_list に置き換えたものとなる。

方法としては，通常の述語と同じように $A_1, A_2, \ldots$ を順番に使用するものが考えられる。この場合，リスト L の N 番目の要素 E を求める述語 nth(N,L,E)，即ち；

```
nth(1,[E|_],E):-!.
nth(N,[_|L],E):- N1 is N - 1, nth(N1,L,E).
```

の第二クローズは，以下のようにコンパイルされる。

```
                              % nth(N,
get_list        A2            %       [
unify_void      1             %        _|
unify_variable  X4            %          L],
get_variable    X5, A3        %             E):-
                              %   subtract(N,
put_integer     1, A2         %              1,
subtract
get_variable    A1, A3        %                N1),
                              %   nth(N1,
put_value       X4, A2        %         L,
put_value       X5, A3        %           E)
execute         nth           %             .
```

このコードの問題点は，subtract の引数設定のために $A_1 \sim A_3$ が使用されるため，L と E に関する最適なレジスタ割当ができないことにある。即ち，これらの変数を A_2 と A_3 に割当てて，get_variable や put_value を除去することができない。

そこで，組込述語の引数は任意の引数レジスタ経由で受渡す方式を考案した [Nakashima 86, 87c]。この方式では，組込述語に対応する機械命令が引数レジスタ番号をオペランドとして持つ。従って，組込述語を処理するマイクロプログラムはオペランドの抽出を行わなければならないが，この操作は通常の機械命令のために用意されたオペランド・フィールドによる引数レジスタのアクセス機構を利用すれば簡単に実現することができる。

この方式を用いると，前述の nth の第二クローズは以下のようにコンパイルされる。

```
                              % nth(N,
get_list          A2          %       [
unify_void        1           %        _|
unify_variable    A2          %          L],
                              %             E):-
                              %   subtract(N,
put_integer       1, X4       %                1,
subtract          A1, X4, A1
                              %                  N1),
                              %   nth(N1,
                              %           L,
                              %             E)
execute           nth         %               .
```

これを単純な方式のものと比較すると，L と E に関する最適割付の結果，get_variable が一つと，put_value が二つ除去されたことが判る。更に，N1 に関しても最適割付が行われ，get_variable がもう一つ取り除かれている。従って合計 4 命令が除去されたこととなり，処理速度とコード量の両面で高い最適化効果が生じている。

更に，subtract に関しては，第二引数を命令のオペランドに埋め込んだ，'subtract_constant Ai,N,Aj' を用意して，put_integer を取り除いている。この命令の導入によって、単に put_integer が除去されるだけではなく，組込述語処理の中での第二引数のデレファレンスやタイプ・チェックが不要となるという効果もある。このタイプの命令は他の算術演算に関しても用意されているが，第二引数がレジスタのものに比べて 30 倍以上の頻度で実行されることが明らかになっており，その効果は非常に大きいものであることが確認されている。

4.2 バックトラックの最適化

バックトラックは論理型言語の大きな特徴の一つであり，また重要な制御機構でもあるが，その「一般的」な処理はかなり「重たい」ものである。即ち，バックトラックを実現するためには，OR ノード通過時点での状態の保存と，ユニフィケーション失敗時の状態の復元が必要であるが，これは従来の計算機におけるコンテクスト・スイッチに匹敵する処理である。しかも，論理型言語においてはあらゆる条件分岐がバックトラックを用いて実現されるため，その実行頻度はかなり高いものとなる。従って，この「複雑で頻度が高い」というバックトラックの性質を，何らかの方法で改善しない限り，論理型言語の高速実行は望めない。

さて，バックトラックに関する処理には以下のものがある。

(1) クローズの選択
(2) 非決定的に選択された場合の Choice Point の生成
(3) ユニフィケーション失敗時のバックトラック
(4) カットと Tidy Trail
(5) 未定義変数への代入時のトレイルとその要否の判定

まず，(1) に関しては 2.4 で述べた Clause Indexing が大きな効果を発揮する。即ち，試行対象となるクローズの数が少なくなることによるバックトラック回数が減少するだけではなく，クローズが決定的に選択される確率が高くなり，(2) の Choice Point 生成の省略による高速化が期待できる。但し，2.4 で述べた引数のタイプと定数／構造体のハッシュ値によるものだけでは，決定的選択の確率やクローズ選択に要する時間に問題がある。

次に，Clause Indexing では決定的選択ができなかった場合，(2) の Choice Point 生成と (3) のバックトラックが行われるが，実はこれにも「簡単な」ケースは存在する。即ち，ヘッド・ユニフィケーションの失敗により同じ述語の別のクローズに移行する，所謂シャロー・バックトラックでは，バックトラック時点での状態と Choice Point に保存された状態の差異が少ないため，必要な情報のみを復元することによりバックトラックに要する手間を削減することができる。また，シャロー・バックトラックしか起こりえないことが判っていれば，Choice Point に保存する情報も削減することができ，更に高速化できる。

この，「シャロー・バックトラックしか起こりえない」ということは，':-' の直後でのカット，即ち Neck Cut の有無により判断することができる。また，4.1.1 で述べた組込述語の性質を用いると，「':-' に引続く組込述語の後のカット」も Neck Cut とみなすことができる。なお，Neck Cut は結果的に「何もしない」ことになりやすいことや，Tidy Trail が単純化できるなど，最適化の対象となりうる操作が数多く存在している。

最後のトレイルに関しては，どのような最適化を行っても要否判定は必須となる。しかし，要否判定の方式自体に高速化の余地がある。

以下本節では，上記に関する独自の最適化手法，即ち Clause Indexing の拡張， Neck Cut とシャロー・バックトラックの高速化，及びトレイル要否判定に関する工夫について述べる。

4.2.1 Clause Indexing

2.4 で述べた Clause Indexing，及び 3.2.2 で述べた WAM の Clause Indexing 用の命令には，二つの問題点がある。一つは定数や構造体による Indexing にハッシュ法のみを用いていることであり，もう一つは第一引数が変数であるクローズを決定的に選択できないことである。

第一の問題点は，クローズの数が小さい時に顕在化する。例えば；

```
p(a):- ...
p(b):- ...
```

のような述語に対しては，ハッシュ法よりも順次探索のほうが普通は高速である。また，アトムのハッシュ値としてはアトム番号の下位ビットが通常は用いられるが，a と b の下位ビットが一致していることは少なからずある。このような場合にはハッシュ表を大きくするか，または衝突させて順次探索をしなければならない。たとえば最下位ビット（ビット0）が一致しているとすると，ビット1が不一致であったとしても衝突を避けるためにはハッシュ表は4エントリ必要であり，ビット1が一致すれば更に大きなハッシュ表が必要となる。もちろん，不一致のビットを対象とするようなハッシュを行うことも可能であるが，それによってハッシュの手間はさらに増加する。

そこで，ハッシュ法ではなく順次探索によってクローズ選択を行うための命令，'jump_on_constant Ai,C,Lab を導入した。この命令は A_i が C に等しければ Lab に分岐する単純な条件分岐命令であり，前述の例については；

```
jump_on_constant  A1, a, Clause_1
jump_on_constant  A1, b, Clasue_2
```

とコンパイルされる。また，この命令はハッシュ値が衝突した時にも用いられる。即ち，ハッシュ法による Clause Indexing のための命令 'hash_on_value Ai,Table' の Table には，衝突が起こっていなければクローズのアドレスが，衝突が起こっていれば jump_on_constant の列のアドレスが，それぞれ格納される。

なお，構造体（ベクタ）に関する Indexing のためには，命令 encode_vector Xn,Ai が用意されている。この命令は A_i が指すベクタの要素数を n，また第一要素である構造体名のアトム番号を a とした時；

$$k(a,n) = n \cdot 2^{24} + a + n$$

で表される値 k を X_n にセットする。これは付録 A に示す ESP のメソッドのためのハッシュ・キー生成法と同じであり，名前が同じで要素数が異なる構造体に，異なるキーとハッシュ値を与えるための工夫である。これを用いて，例えば；

```
p(f(X),...):- ...
p(f(X,Y),...):- ...
p(g(X),...):- ...
```

のための Indexing は；

```
encode_vector      Xj, A1
jump_on_constant   Xj, k(f,1), Clause_1
jump_on_constant   Xj, k(f,2), Clause_2
jump_on_constant   Xj, k(g,1), Clause_3
```

のように行われる。また，クローズが多い場合には；

```
encode_vector  Xj, A1
hash_on_value  Xj, Table
```

とコンパイルされ，定数による Indexing と統一的に取扱われる。

もう一つの問題点である，第一引数が変数であるクローズに関する好例として，前述の nth がある。即ち；

```
nth(1,[E|_],E):-!.
nth(N,[_|L],E):- N1 is N - 1, nth(N,L,E).
```

は，3.2.2 で述べた手法によれば；

```
nth:    try_me_else      C2
        switch_on_term   A1, C1a, L1, fail, fail
L1:     switch_on_constant  A1, <1:L2>
L2:     retry            C1
        jump             C2a

C1a:    retry_me_else    C2a
C1:     "codes for nth(1,[E|_],E):-!."

C2a:    trust_me
        "codes for nth(N,[_|L],E):- ..."
```

とコンパイルされる。即ち，第二クローズの第一引数が変数であるため，try_me_else によって無条件に Choice Point が生成される。従って，N が 1 より大きい間は；

try_me_else → switch_on_term → switch_on_constant → *backtrack* → trust_me

の順で，かなり手間のかかる処理が行われる。

さて，nth の意味を良く考えると，第一引数が「1 ではない」時には第二クローズを決定的に選択して良いことが判る。即ち，第一引数が「一致する」クローズではなく，「一致しない」クローズを選択するという発想の転換を行えば，nth の実行のほとんどを占める第二クローズの選択が決定的となる。つまり，Clause Indexing は「試行できるクローズを選択する」のではなく，「試行することが無意味なクローズを除去する」操作であると考えれば良い。

この考え方に基き，前述の jump_on_constant を用いると，nth は以下のようにコンパイルされる*。

```
nth:    switch_on_term   A1, C1a, L1, C2, C2
L1:     jump_on_constant   A1, 1, L2
        jump               C2
L2:     try                C1
        trust              C2

C1a:    try_me_else    C2a
C1:     "codes for nth(1,[E|_],E):-!."

C2a:    trust_me
C2:     "codes for nth(N,[_|L],E):- ..."
```

即ち，第一引数が 1 より大きい時には；

switch_on_term → jump_on_constant → jump → C2

の順で実行され，Choice Point 生成やバックトラックは発生しない。

更に，このようなパターンでのループはかなり出現すると予測されるため；

jump_on_non_unifiable_value Ai, C, Lab

という命令を用意している。これは A_i の値と C がユニファイできない時，即ち A_i が未定義変数でも C でもない時に，Lab へ分岐する命令である。これを用いると nth は；

```
nth:    jump_on_non_unifiable_value     A1, 1, C2

C1a:    try_me_else     C2a
C1:     "codes for nth(1,[E|_],E):-!."

C2a:    trust_me
C2:     "codes for nth(N,[_|L],E):- ..."
```

とコンパイルされ，さらに高速化される [Nakashima 86]。

*L2 における try と trust を除去し，jump_on_constant の分岐先を C1a とすることができるが，そのような最適化は本題ではない。

4.2.2 Neck Cut Optimization

バックトラックに関する処理の最適化に深く関与するのが，Neck Cut，即ち；

':-' またはそれに引続く組込述語の直後に出現するカット

である。

まず，Neck Cut は；

- 実行中のクローズが Alternative を持っていなければ何もしない
- Alternative を持っていれば最新の Choice Point を除去する

という処理であり，これは除去すべき Choice Point の発見が容易であるなど，他のカットに比べて「簡単な」処理である。また，[Tateno 89] での評価ではカットの中の 70% 以上を占めており，最終クローズ以外でのカットの出現頻度が約 55% であることを考え合わせると，非常に高い頻度で実行される。従って，「簡単でかつ高頻度」という最適化の基本条件を満たしている。

また Neck Cut は，やはり「簡単で高頻度」の処理であるシャロー・バックトラックとも深く関わっている。即ち，最終クローズを除く全てのクローズが Neck Cut を持っているような述語に関しては，それに対応する Choice Point がシャロー・バックトラックによってのみ参照されることが事前に判る。このような場合には，Choice Point に保存する情報を減らすことが可能となり，高速な処理を行うことができる。また，このような述語における Neck Cut では，Tidy Trail の処理も簡単化できる。

以下，Neck Cut に関するこれらの最適化，即ち Neck Cut 自身の高速化と，シャロー・バックトラック及びその特殊例や Tidy Trail の最適化について述べる。

4.2.2.1 Neck Cut 自体の最適化

Neck Cut の処理は前述のように；

(1) 実行中のクローズが Alternative を持っていなければ何もしない
(2) Alternative を持っていれば最新の Choice Point を除去する

というものである。これを実現するには，まず (1) と (2) のどちらのケースであるかを判別しなければならない。

さて，そもそもカット処理は OR 分枝を明示的に除去するものであるから，(1) のような「無意味な」カットは例外的なものであるように見える。しかし，[Tateno 89] での評価によれば，無意味なカット処理が実に 40% 以上を占めている。この事実は，Clause Indexing と深く関わっている。例えば；

```
p(a):-!, ...
p(b):-!, ...
p(c):- ...
```

を，引数を a または b として呼出した場合，Clause Indexing によって第一または第二クローズが決定的に選択されるため，これらのクローズのカットは何もしないこととなる。しかし，これらのカットが不要となるのは，Clause Indexing という言わば処理系の「都合」によるものであり，基本的にはプログラマの関知するところではない。また，筆者などは Clause Indexing が行われることを「知っている」が，OR 分枝の除去を「明示する」ために必ずカットを付けたプログラミングを行う。従って，プログラマにこのようなカットを「除去せよ」ということには無理がある。

このように，(1) のケースの頻度は比較的高く，また「何もしない」という極めて簡単な操作であるため，最適化の対象として選択すべきものであるといえる。そこで，クローズが決定的に選択されたことを示すフラグ **DET** を用意して，(1) と (2) の判別を高速に行う最適化を考案した [Nakashima 86, 87c]。

まず，**DET** のオン／オフは以下のように行われる。

- ゴールの呼出の際に `call` や `execute` によってオン
- Chocie Point 生成の際に `try(_me_else)` によってオフ
- 最終候補節の選択の際に `trust(_me)` によってオン

また，**DET** は Neck Cut を行う命令 `cut_me`* により参照され，オンならば何もせず，オフならばカット処理を行う。なお，**DET** はハードウェアのスイッチ・フラグを用いて実現しているため，そのオン／オフ操作は他の操作と並行に行われ，オーバヘッドは一切ない。また，`cut_me` における判定も高速であり，「何もしない」時の実行サイクルは PSI-II では 1，PSI-III では 2 と，極めて短い時間で実行される。このような効果は **DET** の導入によりもたらされたものであり，2.3 で述べた [Debray 86] のようにレジスタなどにカット後の最新 Choice Point をセットする方法よりも優れている。

一方 (2) のケース，即ちカット操作が必要な場合も，Neck Cut は他のカットに比べて高速な処理を行うことができる。即ち，除去すべき Choice Point がレジスタ **B** が指示する最新の Choice Point のみであることが明らかなため，カット後に最新となる Choice Point の発見やローカル・スタックの縮退は極めて容易である。また，Tidy Trail が真に必要な範囲が，除去される Choice Point が保持する **TR** により明らかになるため，トレイル・スタックの無用な検索も回避することができる。

*Neck Cut 以外のカットは命令 `cut_normal` によって行われる。

4.2.2.2 シャロー・バックトラックの最適化

ヘッド・ユニフィケーションまたは ':-' 直後の組込述語の失敗により，次のクローズへ移行するシャロー・バックトラックは，バックトラックの中でも頻度の高い処理である。例えば，リストLの中に要素Eが存在するか否かを調べる member(L,E)，即ち；

```
member([E|_],E):-!.
member([_|L],E):- member(L,E)
```

では，Clause Indexing は全く行われず，第一クローズのヘッド・ユニフィケーションの失敗によるシャロー・バックトラックが，ループを一回廻るたびに行われる。また，*Quick Sort* の一部である partition，即ち；

```
partition([],_,L,L):-!.
partition([X|L1],Y,[X|L2],L3):- X < Y, !, partition(L1,Y,L2,L3).
partition([X|L1],Y,L2,[X|L3]):- partition(L1,Y,L2,L3).
```

では，XがYよりも小さくない時にシャロー・バックトラックが行われて，第三クローズが選択される。

一般のプログラムにおいても，特に ':-' 直後の組込述語の失敗によるシャロー・バックトラックが頻発する。例えば，[Nakajima 86a, 86b] ではバックトラック全体の 60 ～ 98% を占めることが，また [Carlsson 89] でも 50 ～ 80% であることが，それぞれ報告されている。

さて，シャロー・バックトラックは「生成したばかりの」Choice Point を用いた状態の復元操作であるので，復元前後の状態の差異が比較的少ないという性質がある。まず単純には，Choice Point に保存されバックトラック時に復元されるレジスタ，即ち **E**, **CP**, **G**, **TR**, **LVLC**, $\mathbf{A_1} \sim \mathbf{A_n}$ の内，**CP** と **LVLC** は変化しない。また，ヘッド・ユニフィケーション完了後に allocate を行うようにすれば，**E** も変化しない。更に，引数レジスタの最適割付によるレジスタ間転送の除去をあきらめ，ヘッド・ユニフィケーション完了までは引数レジスタを破壊しないようにすれば，$\mathbf{A_1} \sim \mathbf{A_n}$ も変化しない。従って，バックトラック時にシャローか否かの判断を行い，シャローならば復元が必要な **G** と **TR**，及び候補節のアドレスだけを Choice Point から読出すようにすれば，かなりの高速化が期待できる。

これを更に進めて，最初から Choice Point に保存する情報をシャロー・バックトラックの際に必要なもののみに限定することが考えられる。この最適化の実現には，以下に示す二つの異なる方法がある。

(1) シャロー・バックトラックしか起こりえない時，即ち最終クローズ以外のクローズ全てに Neck Cut がある場合に，限定的な Choice Point を生成し，それ以外の場合には通常の Choice Point を生成する [Nakashima 86, 87c]。

(2) とりあえず限定的な Choice Point を生成し，最初の通常ゴールを呼出す時点でカットされていなければ，通常の Choice Point に置き換える [Yokota 84, Carlsson 89]。

この両者の得失を判断するのは極めて難しいが，筆者らは以下の根拠によって (1) を選択した。

(a) 適用範囲の広さでは (2) が勝っているが，シャロー・バックトラックの大半は Neck Cut を伴う述語で発生するものと期待できる。実際，[Tateno 89] での評価によれば，Choice Point 生成を要する述語の約 2/3 が最適化対象となっており，適用範囲は充分に広いことが明らかになっている。また，適用範囲が真に問題になることが明らかになれば，後述する方法によって一般の述語も対象とするように拡張することができる。

(b) (1) の場合には通常の Choice Point への変換がないことを前提に*，限定的 Choice Point をレジスタに置いているが，(2) の場合にこれを行うと変換時にオーバヘッドとなる。これを避けるために [Carlsson 89] では限定的 Choice Point をローカル・スタックに置き，変換の際には必要な情報の「積み増し」を行っているが，レジスタに置く方法よりも性能が悪い。また，4.2.2.3 で述べる Tidy Trail の最適化を行うとさらに変換時のオーバヘッドが大きくなる。

さて，この最適化のために必要な命令は，fast_try_me_else と fast_try の二つである。これらの命令は，保存が必要な **G** と **TR** に関して；

- $\mathbf{G} \rightarrow \mathbf{GB}$; $\mathbf{GB} \rightarrow \mathbf{BG}$
- $\mathbf{TR} \rightarrow \mathbf{BTR}$

の退避を行う。また，トレイル処理のために **B** をその時点でのローカル・スタック・トップにセットする必要があるため；

$$\mathbf{B} \rightarrow \mathbf{BB} \ ; \ \mathbf{L} \rightarrow \mathbf{B}$$

を行う。更に，候補節のアドレスをレジスタ **AP** にセットし，最新の Choice Point が限定的なものであることを示すフラグ **F_MODE** をオンにする。なお，従来の try(_me_else) には，この **F_MODE** をオフにする操作が付加される。

バックトラック発生時には **F_MODE** を判定し，その値に応じた復元操作を行う。また，バックトラック後に実行される retry(_me_else) と trust(_me) も **F_MODE** に応じた処理を行う。即ち retry(_me_else) では **AP** の更新のみを，また trust(_me_else) では **F_MODE** のクリア，及び **B** と **GB** の復元を行う。

*例外的には通常の Choice Point への変換が発生する。4.3.2 参照。

また，cut_me も F_MODE に応じた処理を行い，オンであれば F_MODE のクリア，B と GB の復元，及び BTR からトップ側の Tidy Trail を行う。なお PSI-III では，F_MODE の値によって命令処理マイクロプログラムの実行開始番地を変更する機構（3.3.2 参照）を用いて，パイプライン化による判定処理の高速化を図っている。

なお一般の述語，即ち Neck Cut を持たないクローズが存在する場合に，部分的に最適化を適用することは可能である。即ち，Neck Cut を持つクローズの列*の先頭と末尾で fast_try(_me_else) と trust(_me) を，また Neck Cut を持たないクローズ列の先頭と末尾で try(_me_else) と trust(_me) を，それぞれ実行するようにすれば良い。しかしこのような拡張がどれほどの効果があるかは疑問であり，またコンパイラの処理が複雑化することもあって，現時点では実施していない。

この他，ヘッド・ユニフィケーション及び引続く組込述語の実行時に，引数レジスタを破壊しないようにコンパイルすることによる性能低下にも，配慮しなければならない。特に，実際は Clause Indexing によって Choice Point 生成やバックトラックが起こらないような述語に関しては，この性能低下は腹立たしいものがある。現時点では最終クローズに関してのみ，ヘッド・ユニフィケーションなどで引数レジスタを破壊するコードを生成している。この配慮と，ループのための再帰呼出が最終クローズで行われることが一般的であることから，ほとんどのループの処理に関しては問題がないとは考えられる。しかし；

```
append([X|L1],L2,[X|L3]):-!, append(L1,L2,L3).
append([],L,L).
```

という「変則的な」プログラミングをするプログラマが皆無であるとは言い切れないし，switch-case のような処理では最終クローズの選択確率が高いとは言えない。従って，最適化の適用範囲をもう少し限定し，例えば；

> Clause Indexing の結果，第一引数が未定義変数でなければ必ず決定的なクローズ選択が行われる時には，最適化を行わない。

というような規則の追加が必要かも知れない。

4.2.2.3 Tidy Trail の最適化

シャロー・バックトラックの最適化は，Choice Point の生成とバックトラックには大きな効果があるが，選択するクローズが決定した場合に行われる Neck Cut の処理は，さほど高速化されない。この主な理由は，かなり複雑な処理である Tidy Trail にあるため，これを除去する方法を考案した。

*ソース・プログラム上は連続していないクローズが，Clause Indexing により連結される場合も含む。

まず，シャロー・バックトラックによってクローズ選択を行っている間は，ほとんど全ての変数代入がトレイルされると言う性質がある。実際，トレイルが不要な変数代入は，ヘッド・ユニフィケーションで生成した構造体の要素を，別の引数のユニフィケーションを経由して代入する場合に限られる。例えば；

```
p([X|Y],X,a):- ...
```

をゴール p(Z,W,W)（Z と W は未定義変数とする）で呼出した場合の，第三引数のユニフィケーションがトレイル不要な代入の例であるが，このようなユニフィケーションは極めて稀であると考えられる。従って，全ての変数代入をトレイルしてもほとんど無駄ではないということができる。

この事実に基き，最新の Choice Point が限定的なものである時にはトレイル要否判定の意味を変更し，Neck Cut によって除去されるか否かによってトレイルする場所を変える方法を提案した [Nakashima 87c]。この方法では，限定的 Choice Point の生成時に B と GB を更新せず，これらをトレイルする場所の選択に用いる。例えば，ローカル・スタックに関する通常のトレイル処理が；

```
if (X < B) push_trail(X) ;
```

であるのに対し，F_MODE がオンである時には；

```
if (X < B) push_trail(X) ;
else       push_special_trail(X) ;
```

とする。即ち，B と GB は必ず行われるカットの後で最新となる Choice Point に対応しているため，Tidy Trail で除去されないものは普通のトレイル・スタックに，除去されるものは特別なトレイル・スタックにプッシュされる。

この方法により，カットの処理は極めて単純化される。即ち Tidy Trail は「特別トレイル・スタック」を空にするだけの操作であり，事実上何もする必要がない。また B と GB が更新されないため，その復元操作も不要となる。一方，バックトラックの際には「特別トレイル・スタック」とトレイル・スタックの双方を用いた Undo が必要であるが，Undo の回数自体は極めて多くの場合変わりがない。また，trust(_me_else) における B と GB の復元操作も不要となる。

しかしこの方法では，様々なユニフィケーション命令で行われるトレイル処理を，F_MODE のオン／オフによって変更する必要がある。これを，通常のトレイル処理の性能を低下させることなく実現することは極めて困難であることが明らかとなり，実際には採用されなかった。

そこで，「無条件にトレイルする」という性質を用いて，変数代入のためのメモリ書込の際にそのアドレスを記憶する「トレイル・バッファ」を考案し，PSI-III においてこれ

を実装した。トレイル・バッファは16wのスタックであり，特別なメモリ書込コマンド unify_write が実行されると，メモリ書込と並行してそのアドレスがプッシュされる。この unify_write は変数代入のための書込のために用いられるが，**F_MODE** に無関係に実行される。即ち，ユニフィケーション命令は **F_MODE** のオン／オフに関わらず unify_write を行い，かつトレイル処理も **B** と **GB** を用いて通常どおり行う。但し，**F_MODE** がオンの場合には **B** と **GB** がカット後に最新となる Choice Point に対応するので；

- Tidy Trail によって除去されないものはトレイル・スタックのみに
- 除去されるものはトレイル・スタックとトレイル・バッファの双方に

それぞれプッシュされることとなる。従って，トレイル・スタックへのプッシュ自体がかなり減少すると言う効果が，副次的なものではあるが期待できる。なお，**F_MODE** がオンでない時にはトレイル・バッファには意味のない値がプッシュされるが，結局参照されることがないため全く問題にはならない。

さて，fast_try(_me_else) ではトレイル・バッファに「意味を持たせる」ために，トレイル・バッファのトップ・ポインタをボトムに設定するコマンド clear_trail_buffer を実行する。この操作は，バックトラック時にトレイル・バッファのトップ・ポインタをコマンド get_trail_buffer_top によって知り，その値を Undo 操作の回数として利用するために行う。即ち，トレイル・バッファには Tidy Trail には無関係に変数アドレスが保存されているので，**F_MODE** がオンである時の Undo は，トレイル・バッファのみを対象として行えば良い*。更に，トレイル・バッファをポップしてそのアドレスに未定義変数セルを書込むコマンド undo が用意されているため，通常の「変数アドレス読出 + 未定義変数セル書込」よりも高速に実行される。

一方，**F_MODE** がオンである時の cut_me は，**F_MODE** をオフにする以外には何もしなくて良い。即ち，トレイル・スタックは既に「Tidy Trail 完了後」の状態になっており，また **B** や **GB** の復元操作も不要である。

なお，PSI-Ⅲ のトレイル・バッファは PSI-I のものとは異なり，アンダーフローはもちろん，オーバフローの心配も不要となっている。即ち，コンパイラはヘッド・ユニフィケーションによって行われる変数代入の回数を予測し，その値がトレイル・スタックの深さである 16 以下である時にのみ fast_try(_me_else) を用いた最適化を行う。従って，cut_me に到達するまでにトレイル・スタックがオーバフローすることはありえない。なお，ヘッド・ユニフィケーションの中に，get/unify_value による「一般的な」ユニフィケーションがあると，コンパイラは完全に変数代入の数を予測できない。即ち，それが構造体間のユニフィケーションであると，任意回の変数代入が発生する可能性がある。しかし，構造体間のユニフィケーションは稀であるため [Touati 87]，一定の数† を get/unify_value に割当

*トレイル・スタックに関しては **TR** に **BTR** をセットして元に戻す操作のみとなる。

†1 で充分である

て，実行時にこれを超える時には 4.3.2 に示す例外的事象として，通常の Choice Point への変換を行えば良い。また，コンパイラの予測値自体が 16 を超えることはさらに稀であり，最適化の範囲が狭まることは全くといって良いほどない。

4.2.2.4 if-then-else

シャロー・バックトラックの最適化を行っても保存と復元が必要なのは，ヘッド・ユニフィケーションによって更新される可能性がある **G** と **TR** である。しかし，もしこれらの値が変わらないことが保証できれば，保存と復元が不要となるばかりではなく，Undo も行われることがないため，バックトラックを単なる分岐とすることができる。これが保証できる述語は，最後のクローズを除いて；

- Neck Cut を持つ。
- ヘッド引数と Neck Cut までの組込述語の出力引数が全て異なる変数である。
- Neck Cut までの組込述語が構造体生成などグローバル・スタックを伸ばす操作をしない。

というクローズからのみ構成されているものである。

このようなクローズは極めて稀なように思われるが，「クローズ内 OR」の存在によって出現することがしばしばある。例えば，partition をクローズ内 OR を用いて記述すると；

```
partition([],_,L,L):-!.
partition([X|L1],Y,L2,L3):-
        (X < Y, !, L2 = [X|L21], partition(L1,Y,L21,L3) ;
                   L3 = [X|L31], partition(L1,Y,L2,L31)).
```

となる。この第二クローズはコンパイラによって概念的には；

```
partition([X|L1],Y,L2,L3):- dummy_predicate(X,Y,L1,L2,L3).

dummy_predicate(X,Y,L1,L2,L3):- X < Y, !,
        L2 = [X|L21], partition(X,Y,L21,L3).
dummy_predicate(X,Y,L1,L2,L3):-
        L3 = [X|L31], partition(X,Y,L2,L31).
```

と変換されるが，この dummy_predicate はまさしく前述の条件を満足している。

このような述語に関するシャロー・バックトラックは；

```
if (built-in-predicate) ... else ...
```

のように実行する最適化が可能である。この最適化手法は if-then-else 最適化と呼ばれ，PSI-I において導入されたものである [Takagi 85]。PSI-II や PSI-III でもこの最適化手法を踏襲しており，比較やタイプ判定など 18 種類の組込述語に関して，失敗時に分岐するアドレスを付加した命令を用意している。例えば前述の変形した partition の第二クローズは；

```
        get_list                A1
        unify_variable          X4
        unify_variable          A1
        jump_unless_less_than   X4, A2, L1
        "codes for L2=[X|L21], partition(X,Y,L21,L3)"

L1:     "codes for L3=[X|L31], partition(X,Y,L2,L31)"
```

のように，分岐アドレスを持った less_than 組込述語命令 jump_unless_less_than を用いてコンパイルされ，極めて高速に実行される。

このようなクローズ内 OR の使用は，論理型言語の持つ特質である書き易さや読み易さをかなり損ねるものであり，それを最適化することによって推奨することには若干のためらいがある。しかし，オペレーティング・システム SIMPOS のプログラマなど，性能を最重視するプロフェッショナルなプログラマには必要悪であると考えられる。

4.2.3 トレイル

バックトラックに関するいかなる最適化を行っても，トレイルの要否判定だけは除去することができない。即ち，Clause Indexing による決定的実行や，トレイル・バッファの導入などにより，トレイル・スタックへのプッシュの頻度は削減できるが，プッシュするか否かの判定はいずれにせよ必要である。しかもこの処理は変数の代入のたびに行われるため，出現頻度はかなり高い。

さてトレイルの要否判定は，変数アドレスとバックトラック・ポイントとの比較によって行われるが，この時バックトラック・ポイントとしてレジスタ B と GB のどちらを使用するかをまず判断しなければならない。即ち，変数がローカル／グローバルのどちらのスタックにあるかを，最初に知る必要がある。この判定は通常，二つのスタックが割付けられたアドレスの大小関係を用いて行う。例えば，グローバル・スタックの方が小さいアドレスであるとし，かつ両者がアドレスの順方向に伸びるものとすると，トレイルの要否判定は；

```
if (X < G) { if (X < GB) push_trail(X) ; }
else       { if (X < B) push_trail(X) ; }
```

となる。また，[Warren 83] のように，グローバル・スタックはアドレスの順方向に，ローカル・スタックはアドレスの逆方向に，対向する形で伸びるものとすると；

```
if (X < GB) push_trail(X) ;
else if (X > B) push_trail(X) ;
```

となり，グローバル・スタック上の変数をトレイルする場合には，比較操作が一回だけで済む*。しかし，最も頻度が高いのは「トレイルしない」場合であり，これに関してはいずれにせよ二度の比較演算が必要である。

さて，変数への代入はユニフィケーション処理の一部であり，無条件に代入が行われることはない。即ち，ユニフィケーションの対象が未定義変数であることの判定が必ず行われており，例えば一方が整数であることが判っている場合であっても；

```
switch (X) {
        case int :   ...
        case ref :   ...
        case undef : ...
        default :    ...
        }
```

の四方向分岐が行われることは，2.2 で述べた通りである†。

*判定順序を逆にして，ローカル・スタック上の変数トレイルを優遇することももちろんできる。

†KL0 では更に hook が加わり五方向となる。

そこで，未定義変数がローカル／グローバルのどちらにあるのかを，このタイプ判定の一環として判断する方法を考案した。即ち，未定義変数のためのタグとして；

lundef　ローカル・スタック上の未定義変数
gundef　グローバル・スタック上の未定義変数

の二種類を用意し，多方向分岐の際にこの両者が分離できるようにした。従って，前述のユニフィケーション処理は；

```
switch (X) {
        case int :     ...
        case ref :     ...
        case lundef : unify_write(&X,Data) ;
                      if (X < B) push_trail ; break ;
        case gundef : unify_write(&X,Data) ;
                      if (X < GB) push_trail ; break ;
        default :      ...
        }
```

のようになり，ローカル／グローバルのいずれに関しても，トレイル要否の判定は一回の比較で行われる。なお，未定義変数を生成する際には，どちらのスタックに生成するかがはっきり判るため，タグを異なったものにするのは極めて容易である。また，このローカル／グローバルの判別は，Tidy Trail の際の除去判定にも用いられ，その意味でも高速化に貢献している。

4.3　例外処理の最適化

2.5で述べたように，KL0 には Prolog にはない様々な実行順序制御機能が盛り込まれており，実用的で大規模なプログラムの開発に大きく役立っている。しかしこれらの機能の内，動的な述語呼出を伴うもの，即ち組込述語の例外 (Exception) と Bind Hook に関しては，これまで本章で述べた最適化手法の実現の支障となるものである。

即ち，組込述語をヘッド・ユニフィケーションの一部とみなす FGO の拡張や Neck Cut Optimization は，いずれも「最初の通常述語の呼出前には...」という前提条件に基く最適化であるが，動的な述語呼出がこの前提を崩してしまう。例えば，ヘッド・ユニフィケーションでフック変数への代入が行われると，':-' の直後に変数にフックされた通常述語が「挿入」される。従って，それに引続く組込述語やカットは，「':-' 直後の...」という性質を失い，最適化の適用条件からはずれたものとなってしまう。同様に，組込述語の実行中に例外が検知されると，組込述語が通常述語である例外ハンドラに「置換」されるため，それ以降は「':-' に引続く組込述語の直後の...」という性質がなくなってしまう。

また Bind Hook に関しては，フック変数への代入から述語呼出までの間に「遅延」が存在するため，代入が行われたことをどのようにして知るかと言う問題がある。同じ問題は，コンパイル・コードのデバッグ機能を実現するためのトレース例外の検出にもあてはまる。

さて，これらの機能は便利なものであり，かつ実用的なシステムには不可欠なものであるが，その性能に関してはさほど厳しい要求があるわけではない。例えば組込述語の例外発生原因のほとんどを占めるプログラム・バグによるエラーに関しては，性能を云々されることはありえない。また，算術演算組込述語による数式処理などの例外を積極的に用いたものや，Bind Hook を用いた遅延評価についても，実行される頻度が小さい上，ソフトウェアによる実現と比較して充分に高速であれば，多少の性能の高低は問題とはならない。同様に，コンパイル・コードのデバッグに関しても，基本的にはユーザとの対話処理であるため，インタプリタの性能を大きく上回ることができさえすれば充分である。

従って，これらの機能を実現するに当っては，例外的な事象である「実際に動作した時」の性能が多少犠牲になっても，「通常の処理」の高速化を優先すべきであると言うことができる。そこで，PSI-II 及び PSI-III ではこの方針に基き，例外的事象の検知とそれに対する処理を，通常処理に悪影響を全く与えることなく実現した。以下，各々についてその手法を述べる。

4.3.1 例外の検出

前述のように，ヘッド・ユニフィケーション時にフック変数への代入が行われた場合，フックされた述語の呼出はヘッド・ユニフィケーションが全て完了した後に行われる。従って，何らかの方法でヘッド・ユニフィケーションの終了と，フック変数の代入の有無を知る必要がある。例えば [Carlsson 87] では，フック変数の代入を記憶するフラグを設け，`call`，`execute`，及びカットやインライン展開された組込述語*の処理の際にこのフラグをチェックする方法が提案されている†。しかし，この方法は `call` や `execute` などの処理に負担がかかるだけではなく，それらよりも実行頻度が高いカットや組込述語の性能低下も招くため，効率の良い方法であるとは言い難い。

また，これを改良した方法として，':-' の直後に `end_head` なる命令を挿入し，フック変数の代入をチェックすることも考えられる。この方法は，カットや組込述語での判定が不要な点が大きなメリットであり，[Carlsson 87] よりは確実に高速化できる‡。なお，組込述語の出力引数のユニフィケーションによりフック変数への代入が発生する可能性のある時には，そのユニフィケーションのための *get* 系命令の直後に `end_head` を置く必要があるが，このようなケースは比較的少ないためさほど問題とはならない。しかし，この方法によっても一回の述語呼出に最低一度はフラグの判定を行う必要があり，オーバヘッドが完全には除去されない。

一方トレース例外の検出は，更に問題が大きい。即ち，トレース例外が発生している時には組込述語を含む全てのゴールがトレース例外ハンドラに置換されるため，[Carlsson 87] と同様の方法で例外を検出せざるをえない。従って，Bind Hook とトレース例外の双方を実現するには，[Carlsson 87] に依らざるをえないように見える。

しかし PSI-II や PSI-III では，割込を用いた巧妙な判定手段を導入し，例外検出のオーバヘッドを完全に除去することに成功した [Nakashima 87d]。この手法は，命令と割込の種類によって割込の許可／不許可を決定する，「命令依存型割込マスク」とも言うべき方式に基いている。例えば，フック変数の代入を行うと「フック変数代入割込」をオンにするが，この割込はヘッド・ユニフィケーションに出現する *get*/*unify* 系の命令ではマスクされ，ユニフィケーション完了後の最初の命令実行時に有効となる。また「トレース例外割込」は，`call`, `execute` 及び組込述語命令以外ではマスクされる。また，これらの割込はプロセス切換などを伴うソフトウェア・レベルでの割込みとは異なり，マイクロプログラムによって述語呼出が行われるため，割込処理自体も高速である。

*比較，タイプ判定など使用頻度の高い組込述語。

†[Carlsson 87] には示されていないが `proceed` でもチェックしなければならない。

‡シャロー・バックトラックが発生すると判定回数が増加するように見えるが，':-' を通過した後のシャロー・バックトラックは [Carlsson 88] でも組込述語の処理中に判定が行われるため，結局判定回数は増加しない。

なお，実際のマスク処理もハードウェアではなくマイクロプログラムで行われる。即ち，ハードウェア・レベルでは命令ごとに割込が発生するが，それを処理するマイクロプログラムが命令コードと割込要因を解析して，割込が許可されない時には単に実行を継続する。このマイクロプログラムによるマスク処理は柔軟である反面，トレース例外のように割込が継続的に起こる場合の性能が問題となる。しかし，最悪の場合でもトレース時の性能低下は 10 倍以内であり，インタプリタに比べれば 10 倍以上の性能を維持することができる。また，割込が発生していない「通常」の場合には，もちろん一切のオーバヘッドはない。

4.3.2 動的な呼出

動的な述語の呼出，即ちヘッド・ユニフィケーションにおけるフック変数代入による「挿入」や，組込述語が例外を検出した場合の「置換」は，それ以後のカット，組込述語，及び最初の通常述語が；

「':-' 及びそれに引続く組込述語の直後」

に存在するものであると言う性質を変えてしまう。これによって，以下の最適化手法が影響を受ける。

(1) ヘッド，最初の通常ゴール，及びそれ以前の組込述語の引数としてのみ出現する変数を，一時変数として引数レジスタに割付ける（拡張した）FGO。

(2) Neck Cut (cut_me) における，フラグ **DET** を用いたカット操作の要否判定と **B** を用いたカット操作。

(3) Neck Cut 以前の組込述語での失敗によるシャロー・バックトラックの最適化と，それに関連するカット操作。

まず (1) の FGO に関しては，引数レジスタの退避／復元を動的な呼出の前後に行えばよい。ここで，どの引数レジスタを退避すればよいかが問題となるが，それには有効な引数レジスタを示すフラグ・レジスタ **RVFR** (3.2.4 参照) が用いられる。また，通常ゴールが一つしかなく FGO と TRO を組合わせた最適化が行われた場合，**E** や **CP** を退避する allocate が行われていないため，これらも同様に退避する必要がある。従って，動的呼出時の際には局所変数が引数レジスタに置き変わった特殊な Environment が生成される。また，bind_hook や exception_hook で定義される述語には，命令 end_of_bind_hook_handler または end_of_exception_handler がその末尾に付加され，これらが特殊な Environment から引数レジスタへの復元操作を行う*。

なお，組込述語の例外に関しては，例外ハンドラに組込述語の引数が引渡されるが，出力引数はレジスタであるためハンドラが何らかの値を格納するための場所がメモリ上には確保されていない。そこで，ハンドラの呼出の際に出力引数の各々について大域変数が生成され，これらが例外ハンドラへ引渡されるとともに，特殊な Environment にも大域変数への Reference Pointer が引数レジスタの値として退避される。例えば；

```
p(X,Y):- Z is X + Y, q(Z).
```

は；

*end_of_bind_hook_handler には一回のヘッド・ユニフィケーションで複数のフックされた述語を呼出す操作が付加されている。


```
p:      add     A1, A2, A1
        execute q
```

とコンパイルされる。ここで add が例外を検出すると，変数 Z が「大域化」されて例外ハンドラに引渡されるとともに，A_1 の旧値として大域変数への Reference Pointer が退避される。従って，例外ハンドラが大域変数に何らかの値をユニファイすると，復元後の A_1 はその値を間接的に指示することとなり，結果的にその値が q に正しく引渡される。

次に (2) の Neck Cut の問題に関しては，**EVENT** というフラグを導入して解決している。まず，end_of_xxx_handler はフラグ **DET** をオフにするとともに **EVENT** をオンにして，cut_me に最適化が適用できないことを通知する。この場合の cut_me の処理は，2.3 で述べたナイーブなカット処理，即ち Choice Point と Environment の連鎖をたどって除去対象か否かを一つずつ判定する処理を行う。

なお，**EVENT** は **DET** がオンである時には意味を持たないため，call，execute，trust(_me) が **DET** をオンにする処理や，cut_me が無動作の場合は全く影響を受けない。また，**DET** をオフにする try(_me_else) は **EVENT** もオフにしなければならず，また cut_me も **DET** がオフの時には **EVENT** をチェックしなければならないが，これらは他の操作と並行して行われるため，やはりオーバヘッドはない。

最後のシャロー・バックトラックの問題については，「限定的」Choice Point を「普通の」ものに変換することによって解決される。即ち，**G** と **TR** に関しては **BG** と **BTR** を Choice Point に退避し，それ以外に関しては try(_me_else) と同様にレジスタの退避が行われ，更にトレイル・バッファの内容が全てトレイル・スタックに移される*。また，F_MODE はオフになるため，cut_me の処理は前述の **EVENT** を利用したものとなる。なお，F_MODE がオンの場合に get/unify_value で構造体間のユニフィケーションを行い，コンパイラが予測したトレイル回数に狂いが生じた時も，同様に Choice Point の変換が行われる。

*移動の前にトレイル・スタックはバックトラック・ポイントまで一旦戻されるため，バッファとスタックの双方にあったものが重複することはない。

4.4 複合命令

連続して出現する可能性の高い複数の命令を，一つの命令にまとめてしまう命令の複合化は，基本的な最適化手法の一つとしてしばしば用いられる。この「複合命令」の導入は，単に命令の数を減らしてコード量や命令フェッチの手間を削減するだけではなく，命令間のインタフェースのための操作の削除や，逐次的に行われていた操作の並列化による高速化も期待できる。

PSI-IIやPSI-IIIでも，水平型マイクロ命令や大容量のWCS，更には豊富な命令コードといったハードウェアの特徴を生かして，表4-2に示す複合命令を導入した。なお，表には複合化によって削減されたステップ数（PSI-IIIの値）を示している。以下，各々について簡単に述べるが，実行頻度に関するデータは[Tateno 89]の評価に基く。

(1) `get_list` + `get_variable` + `get_variable`
リスト・セルの分解に用いられる命令であり，carに関しては一時変数に限定し，cdrに関しては一時変数と局所変数の双方を用意している。これは；

```
p([X|Y],...):- q(X,...), p(Y,...).
```

という典型的なリスト処理において，qが組込述語であるものと，そうでないものの双方に対応するためである。なお，このパターンの出現頻度は`get_list`の1/3以上と極めて高い。

表4-2: 複合命令

instruction	gain
`get_list` + `unify_variable` + `unify_variable`	1
`deallocate` + `execute`	1
`deallocate` + `proceed`	2
`execute` + `switch_on_term`	5
`execute` + `deallocate` + `switch_on_term`	3
`call` + `switch_on_term`	1
`put_atom` + `execute_method`	2
`put_atom` + `deallocate` + `execute_method`	4
`put_atom` + `call_method`	2
`cut_me` + `proceed`	4
`cut_normal` + `deallocate` + `proceed`	6

(2) `deallocate` + `execute`
通常のゴールが二つ以上ある時の最終ゴールの呼出は，基本的にはこのパターンで行われる。従ってその出現頻度は当然高く，ゴール呼出の約15%はこの組合せで行われる。なお，`deallocate`が必要な場合の半数以上はこのパターンであるが，`deallocate` + `proceed`というパターンも約1/4を占めるため，この組合せも用意している。

(3) `call`/`execute` + `switch_on_term`
述語の先頭の命令が`switch_on_term`であることが多く，全体の約20%を占めている。そこでこの組合せを用意しているが，分岐の連続と言うパイプライン・マシンが「苦手の」パターンであるため，ステップ数以上の効果が現れていると思われる。

(4) `put_atom` + `call`/`execute_method`
メソッド呼出のほとんどはこのパターンで行われ，実際99.9%以上を占める。速度の面では単に`put_atom`がなくなるだけではなく，メソッド名がアトムであることのチェックが省ける効果もある。

(5) `cut_me` + `proceed`
通常述語の呼出が全くない（拡張された）ユニット・クローズの，約30%は末尾にカットがあり，`cut_me` + `proceed`のパターンとなる。また，通常述語がある場合にも末尾がカット(`cut_normal`)であることがかなりあり，`cut_normal`の全体の半数以上，また`deallocate`が必要な場合の約20%がこのパターンである。そこで，`cut_normal` + `deallocate` + `proceed`も用意している。

第5章
評価

計算機アーキテクチャの研究において，対象とする計算機の性能や特性を評価することが重要な項目であることは言うまでもない。即ち，事前の評価なくしてはアーキテクチャに関する様々な選択肢の中から，何を選ぶかを決定することはできない。また，事後の評価はそれらの選択の正当性を検証するだけではなく，次のステップのための事前評価としても役立つものである。

筆者らが行ったPSI系列の推論マシンの開発においても，事前／事後の評価を充分に行い，アーキテクチャの決定や改良のための重要な指針として利用した。これらの一部については，前二章におけるアーキテクチャや最適化の議論の中で述べたが，本章では特に重要な項目である実行速度，最適化の効果，及びメモリ・アクセスの特性に関して詳しく論ずる。

まず実行速度に関しては，開発した各々の推論マシンの比較を行うとともに，他のマシンとの比較も行って，アーキテクチャが性能に与える影響を議論する。次に最適化の効果については，個々の最適化手法が高速化にどの程度貢献しているかをベンチマーク・プログラムを用いて詳細に解析するとともに，大規模なプログラムに関する評価結果の報告も行う。最後のメモリ・アクセス特性の評価では，性能に与える影響が大きいキャッシュ・メモリのヒット率に関する解析を行うほか，論理型言語の処理におけるメモリ・アクセスの特徴についても議論する。なお，本章に関する研究は，三石彰純，立野裕和，佐伯稔らの協力を得ておこなった。

5.1 実行速度

5.1.1 方式による性能差

PSI-IからPSI-IIIに至る開発の過程において，ハードウェア，マイクロ命令，処理方式，最適化など，様々な面で改良を行ってきた。これらの改良がどのような効果を発揮したかを知るには，同じプログラムの実行時間の差を見るのが最も適当である。そこで，PSI-I, PSI-II, PSI-IIIの三つの推論マシンと，3.2.2で述べたPSI-I上の実験的WAM処理系に関して，いくつかのベンチマーク・プログラムの実行時間を測定した。

5.1.1.1 ベンチマーク・プログラム

ベンチ・マークプログラムは主にProlog Contest [Okuno 84]に含まれるものであり，それらの述語数，クローズ数，及びLogical Inferenceの数は表5-1に示す通りである。以下，各々について簡単に説明する。

append 2つのリストの結合。[Okuno 84]には含まれていないが極めて有名なベンチマークである。決定的なリスト処理が中心である。

nrev リストの反転を $O(n^2)$ の手間で行う，所謂 *Naive Reverse*。Warren Benchmark [Warren 77]の一つであり，要素数は30である。やはり，決定的なリスト処理が中心である。

表5-1: ベンチマーク・プログラム

program	# of predicates	# of clauses	# of inference
append	1	2	1,000
nrev	3	5	498
qsort	3	6	603
trav	2	4	2,124
lisp	9	37	7,421
queen	5	9	5,750
diff	5	26	3,053
solve	4	9	1,302
srev	2	4	854
harmonizer	44	445	16,262

である。

qsort やはりWarren Benchmarkの一つであり，50要素のリストを *Quick Sort* を用いてソートする。シャロー・バックトラックが頻発する。

trav 1000要素のリストの全要素をバックトラックによりたどる。ディープ・バックトラックが頻発する。

lisp Pure Lispのインタプリタで10番目のフィボナッチ数を計算する。基本的には決定的な処理であるが，基本関数の処理でカットのないシャロー・バックトラックが多発する。

queen 所謂 *8-queens* 問題をナイーブなGeneration and Checkで解き，最初に見つかった解を求める。配置のチェックではシャロー・バックトラックが多発し，配置できないと判った時点でディープ・バックトラックがおこる。

diff 記号的な数式処理によって $(x-1)^5$ の5回微分を行う。[Warren 77]のものと基本的には同じであるが，要素が数値である時には実際に計算を行う処理が付加されている。数式の要素が何であるかを知るためのシャロー・バックトラックが多発する。

solve 自然数をSuccessor Function $s(s(\ldots s(0)))$ を用いて表現し，論理型言語の持つ双方向性を利用して逆関数を含んだ式 $fib^{-1}(13)!$ を計算する（$fib(n)$ は n 番目のフィボナッチ数）。カットのないシャロー・バックトラック，ディープ・バックトラック，及び構造体間のユニフィケーションが多発する。

srev 5要素のリスト反転を $O(2^n)$ の手間で行う。カットのないシャロー・バックトラックとリスト間のユニフィケーションが多発する。

harmonizer これはProlog Contestの課題ではなく，FGCS'84で行ったPSI-Iのデモンストレーション用のプログラムであり，与えられたメロディに対して四声の和音を生成する一種のエキスパート・システムである。219種の和音データ・ベースと99種の和音進行規則によるGeneration and Checkが基本であり，カットのないシャロー・バックトラックとディープ・バックトラックがともに多発する。

5.1.1.2 ベンチマーク・プログラムの実行速度

各マシンにおける上記のプログラムの実行時間（実測値）とLIPS値を，表5-2に示す[Okuno 85, Nakashima 87b, 87c, Saeki 91]。また表にはPSI-IとPSI-IIIの速度比を示しているが，PSI-IからPSI-IIIまでの進歩が7～40倍という大きなものであったことがこの値により示されている。なお，この性能向上率がプログラムごとに差があることに関しては，以下のようにプログラムの性質の差を反映したものであると考えられる。

表 5-2: ベンチマーク・プログラムの性能

program	PSI-I	WAM on PSI-I	PSI-II	PSI-III	PSI-I / PSI-III
append	26.4	9.66	2.33	0.66	40.41
	37.5	103.5	430.1	1515.2	
nrev	13.6	4.35	1.36	0.41	33.17
	36.6	114.5	366.2	1214.6	
qsort	15.2	6.73	2.73	0.83	18.31
	40.1	90.5	223.1	733.7	
trav	51.7	28.95	16.14	5.93	8.72
	41.1	73.4	131.6	358.2	
lisp	369.0	*2	102.77	35.88	10.28
	20.1		72.2	206.8	
queen	96.0	48.10	20.93	7.34	13.08
	59.9	119.5	274.7	783.4	
diff	*2	28.28	12.61	5.79	—
		108.0	242.1	527.3	
solve	38.2	22.31	14.04	5.16	7.40
	34.1	58.4	92.7	252.3	
srev	24.8	10.30	6.11	2.34	10.60
	34.4	82.9	139.8	365.0	
harmonizer	656.3	276.00	120.56	60.63	10.82
	24.8	58.9	134.9	268.2	

*1 各欄の上段が実行時間（単位 ms），下段が KLIPS 値。
*2 測定されていない

group-1 ··· *append, nrev*
決定的なリスト処理であり，Clause Indexing, FGO, TRO などの基本的最適化が非常に有効であるため，33 ～ 40 倍の極めて大きな性能比が現れる。

group-2 ··· *qsort, queen*
カットがあるシャロー・バックトラックが多いため，Neck Cut Optimization が有効に作用し，13 ～ 18 倍程度のかなり大きな性能比となる。

group-3 ··· *lisp, srev, harmonizer*
カットのないシャロー・バックトラックが多発するため，最適化の効果がほとんど現れず，性能比は 10 倍程度に留まる。なお *lisp* と *srev* については，本来行うべきカットが欠落していることが主な要因であり，それを修正すると性能比は；

lisp = 17.14
srev = 12.35

に拡大し，group-1 に近づく。

group-4 ··· *trav, solve*
ディープ・バックトラックや構造体間のユニフィケーションのような，最適化がほとんど不可能な複雑な処理が頻繁になされるため，性能比は 10 倍を下回る値となる。

なお，*diff*については PSI-I の測定データがないため比較できないが，プログラムの性質から考えて group-2 に属するものと思われる。

以上の結果をまとめると；

- ディープ・バックトラックや双方向ユニフィケーションなど，Prolog の「論理」の側面を単純に用いた「ナイーブ」なプログラムについてはさほど性能が上がっていないが；
- Prolog を「プログラム言語」として捉えてプログラミングを行い，性能にもそれなりの配慮をしたものについては，大きく性能が向上している；

ということが言える。従って，論理型言語の高速処理のためには，プログラミングに関する一定の指針を樹立することも必要な要素の一つであると考えられる。なお，一般の応用プログラムに関しては，ほとんどが group-2 に，悪くとも group-3 には属しているものと考えられ，PSI-I から PSI-III への性能向上率は 10 倍以上であると思われる。また，PSI-I と PSI-III の論理素子の速度比はほぼ 1:2 であり，アーキテクチャ改良や最適化の効果は 5 倍以上であると見積もられる。

5.1.1.3 インタプリタ方式とコンパイル方式の比較

以下，PSI-I から PSI-Ⅲ までの各ステップでの性能向上について検討する。まず図 5-1に，PSI-I の性能を 1 とした時の，PSI-I 上の WAM 処理系の性能を示す。この二つの処理系は全く同じハードウェアを用いており，性能の差は真にインタプリタ方式とコンパイラ方式の差を表す。また，シャロー・バックトラック以外の最適化は実験処理系にも全て施されており，その効果も現れているものと考えられる。

さて，性能比は 1.7 ～ 3.1 倍であるが，この値は同じハードウェアであること，コンパイル方式では命令のフェッチ／デコードに 1 命令あたり最低 1 サイクルのオーバヘッドがあること，などを考えるとかなり大きなものである。特に，最適化の効果が出にくい group-4 でさえ 1.7 倍以上の性能向上となっていることから，明らかにコンパイル方式が優れていることが証明されている。なお性能比のばらつきは，シャロー・バックトラックの最適化がなされていないためにあまり高速化されていない group-2 以外に関しては，前述の傾向と同じである*。このことから，インタプリタ方式は基本的な処理に要する手間が大きいため，コンパイル方式と同じ思想で行っている最適化が，実際にはあまり効果を発揮していないことが判る。

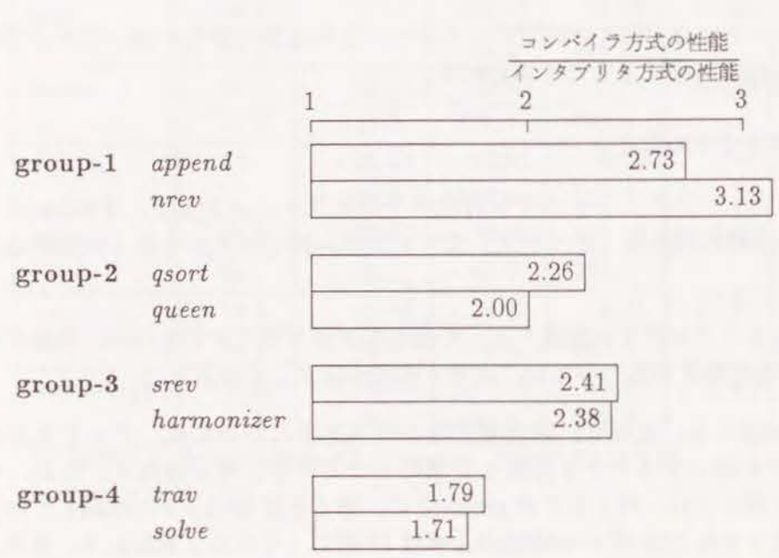


図 5-1: PSI-I におけるインタプリタ方式とコンパイラ方式の性能比

*インタプリタ方式ではシャロー・バックトラックの最適化を行っている。

5.1.1.4 PSI-I と PSI-Ⅱ の比較

まず，PSI-I のインタプリタ方式と PSI-Ⅱ の性能の比を，図 5-2に示す。性能比は 2.7 ～ 11.3 と大きく，またばらつきの傾向は 5.1.1.2 で述べたものと同じである。この二つの処理系は論理素子の速度はほとんど同等であるため，インタプリタ方式からコンパイラ方式への変更と，ハードウェア・アーキテクチャの改良の相乗効果によって，大きな性能向上が得られたものと考えられる。

次に，ハードウェア・アーキテクチャの改良効果を評価するために，同じコンパイラ方式である PSI-I 上の WAM 処理系と PSI-Ⅱ の性能の比を，図 5-3に示す。性能比は 1.6 ～ 4.1 倍であり，改良の効果が大きいことが判るが，性能比のばらつきはインタプリタ／コンパイラの比較結果に比べて大きくなっている。また，PSI-Ⅱ ではシャロー・バックトラックの最適化も導入されており，プログラムによってはこの効果が大きい。

これらを細かく分析すると，性能向上の要因について以下のようなことが考えられる。

(1) データ・バスの負荷や物理的線長の短縮など，ハードウェア・レベルの細かい最適化によるマシン・サイクルの短縮 (200 ns → 155 ns) により，全てのプログラムが最低 1.3 倍高速化される。

(2) 命令フェッチやデコード機構の強化は，一命令あたりの処理ステップが少ない group-

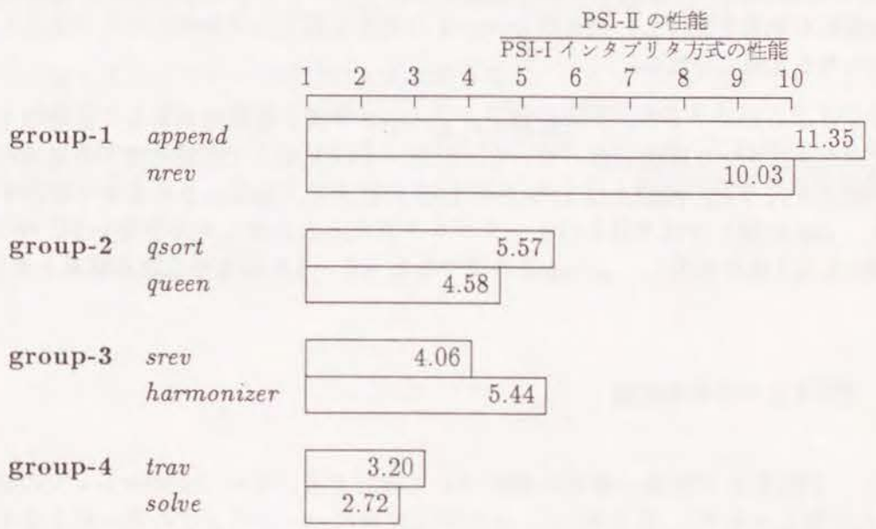


図 5-2: PSI-I （インタプリタ方式） と PSI-Ⅱ の性能比

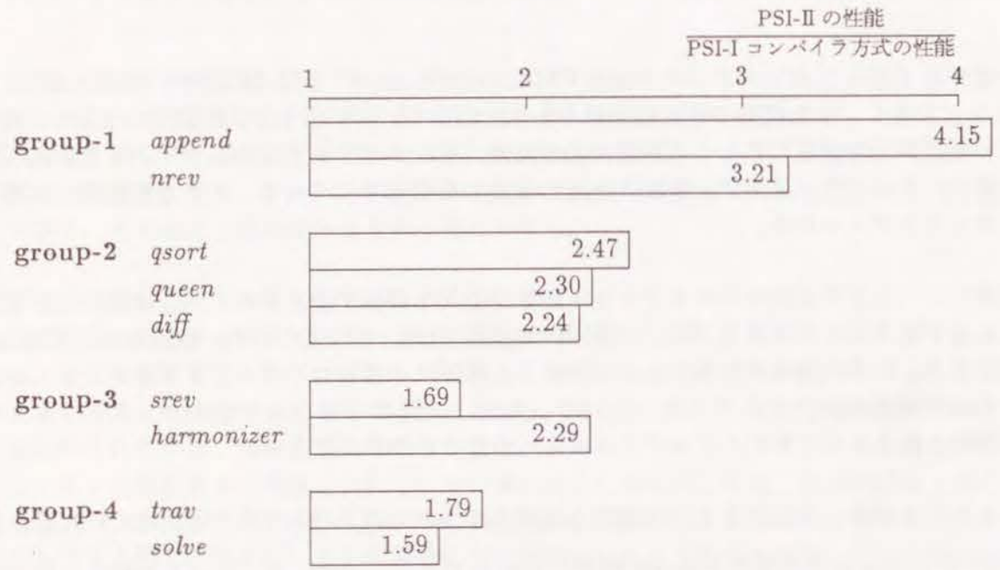


図 5-3: PSI-I（コンパイラ方式）と PSI-II の性能比

1 の高速化に特に効果があり，性能が 3 ～ 4 倍に向上している。

(3) (2) とは逆に，タグに関する分岐機構の強化やマイクロ命令の改善は，比較的複雑な処理にも効果を発揮し，例えば group-4 の高速化要因の半分はこれらの改良によるものであると考えられる。

(4) シャロー・バックトラックの最適化は group-2 で大きな効果があり，2 倍以上の高速化がなされた主な原因となっている。なお，*srev* に対して前述のような最適化のための修正を行うと，性能比は 1.69 から 1.95 に拡大し，group-2 にかなり接近する。また，*lisp* に関しては PSI-I のインタプリタ方式との比較しかできないが，やはり 3.6 倍から 5.7 倍に拡大し，group-2 の値である 4.6 ～ 4.9 倍をも上回る結果となる。

5.1.1.5 PSI-II と PSI-III の比較

最後に，PSI-II と PSI-III の性能の差について検討する。この二つのマシンでの処理方式はほとんど等しいため*，図 5-4に示した性能比は真にハードウェアの違いによるものであ

*シャロー・バックトラックに関するトレイル処理の最適化のみが異なるが，これもハードウェアの違いに深く関係している。

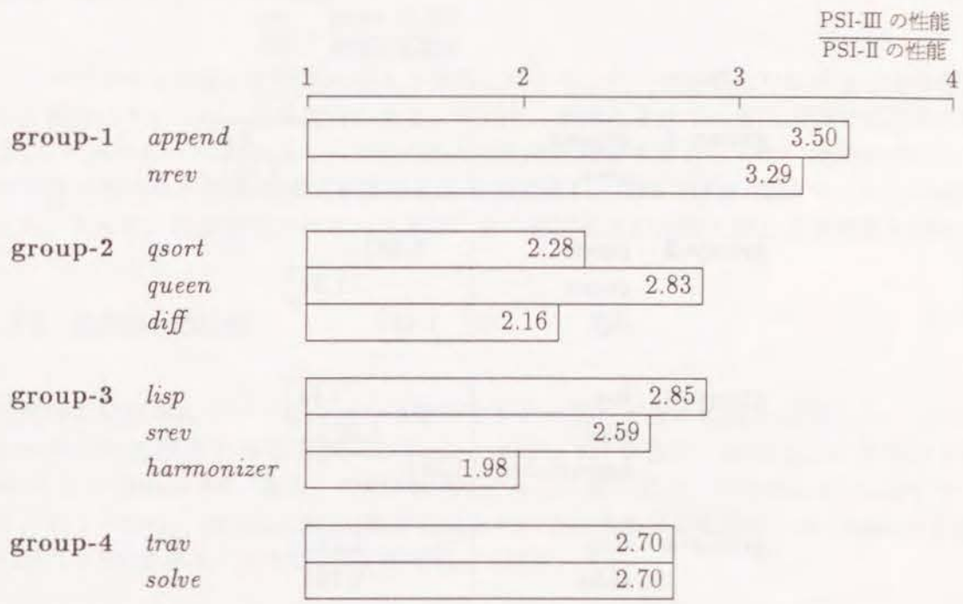


図 5-4: PSI-II と PSI-III の性能比

る。なお，この違いの中には論理素子の速度差が含まれているため，これを正規化した値，即ち PSI-II のマシン・サイクルを 100 ns とした時の性能比を図 5-4に示す。

これらの値を見るとグループ間での大きな差はなく，パイプライン化を中心としたアーキテクチャの改良が万遍なく効果を発揮し，1.5 ～ 2 倍の性能向上をもたらしたものと考えられる。但し，*harmonizer* に関しては向上率がさほど大きくないが，これはコードに関する参照の局所性が余りなく，命令キャッシュ・ミスのペナルティの相対的拡大が悪影響を及ぼしたものと考えられる。

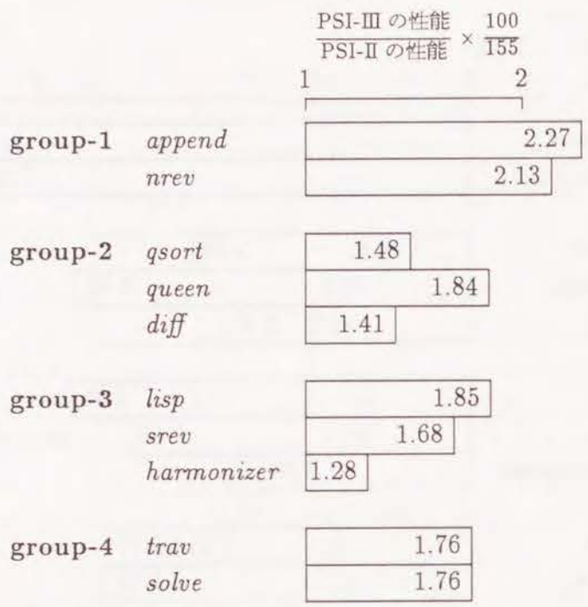


図 5-5: PSI-II と PSI-III の性能比（正規化値）

5.1.2 他のマシンとの比較

アーキテクチャの違いが性能に与える影響を考える上で，汎用機上の処理系や他の専用機との比較を行うことは大きな意味がある。そこで，商用化されている汎用機の処理系に関して 5.1.1 で述べたベンチマーク・プログラムの性能を測定するとともに，商用に近いレベルの完成度を持った専用機の速度を文献によって調査し，PSI 系列の推論マシンとの比較を行った。さらに，最近提案されている RISC 上の処理系との比較に関しても考察を行う。

5.1.2.1 汎用機との比較

汎用機の処理系については，二つの著名なものに関してその性能を測定した。一つは，最初の実用的処理系である DEC-10 Prolog [Bowen 81] であり，使用した計算機は DEC-2060 である [Okuno 85]。なお，DEC-10 Prolog の処理方式は，[Warren 77] に基くものである。もう一つは，[Warren 83] の処理方式を初めて商用処理系に適用した Quintus Prolog [QUINTUS 85] であり，VAX/8700 を使用して測定した。

これらの測定値に基く，PSI 系列の推論マシンとの比較結果を図 5-6に示す。この図には，5.1.1 で述べたベンチマークの各グループを代表するのものとして *nrev, qsort, harmonizer, trav* を選び，それらの性能を PSI-I を 1 として正規化した値を示している。

まず，DEC-10 Prolog の性能は PSI-I の 0.6 ～ 1.4 倍であり，ほぼ同等の性能となっている。逆に言えば，DEC-10 Prolog という当時の最高速処理系と同等の性能を達成すると言う，PSI-I の設計に際しての大きな目標はほぼ達成できたものと考えられる。なお，プログラムの性質による性能比の変化は比較的少ないが，やはり 5.1.1 で述べたものと同じ傾向となっている。

一方 Quintus Prolog は，PSI-I の 0.5 ～ 3.0 倍というかなりばらついた性能比を示し，5.1.1 で述べた傾向が拡大されている。この性質は図 5-7に示した，Quintus Prolog を 1 とした時の PSI-II 及び PSI-III の性能からも読み取れる。これは，PSI-II や PSI-III では「複雑な」処理がマイクロプログラムによって行われるため，同じ WAM 系の処理方式を用いた Quintus Prolog に比べて，ディープ・バックトラックなどのダメージが比較的少ないためであると考えられる。なお，測定に使用した VAX/8700 は約 5 MIPS の性能であると言われており，これに基いて換算すると論理型言語の処理に関する限り，PSI-II は 15 ～ 30 MIPS，PSI-III は 30 ～ 80 MIPS 級の汎用機に相当する性能を持っていると言うことができる。また，他のベンチマーク・プログラムによる測定結果によれば [Saeki 91]，Quintus Prolog を 10 MIPS の性能と言われる SUN4/280 上で実行した性能 [Habata 89] に対して，PSI-II は 1.5 ～ 3 倍，PSI-III は 5 ～ 10 倍の性能であり，前述の MIPS 換算の妥当性が確認されている。

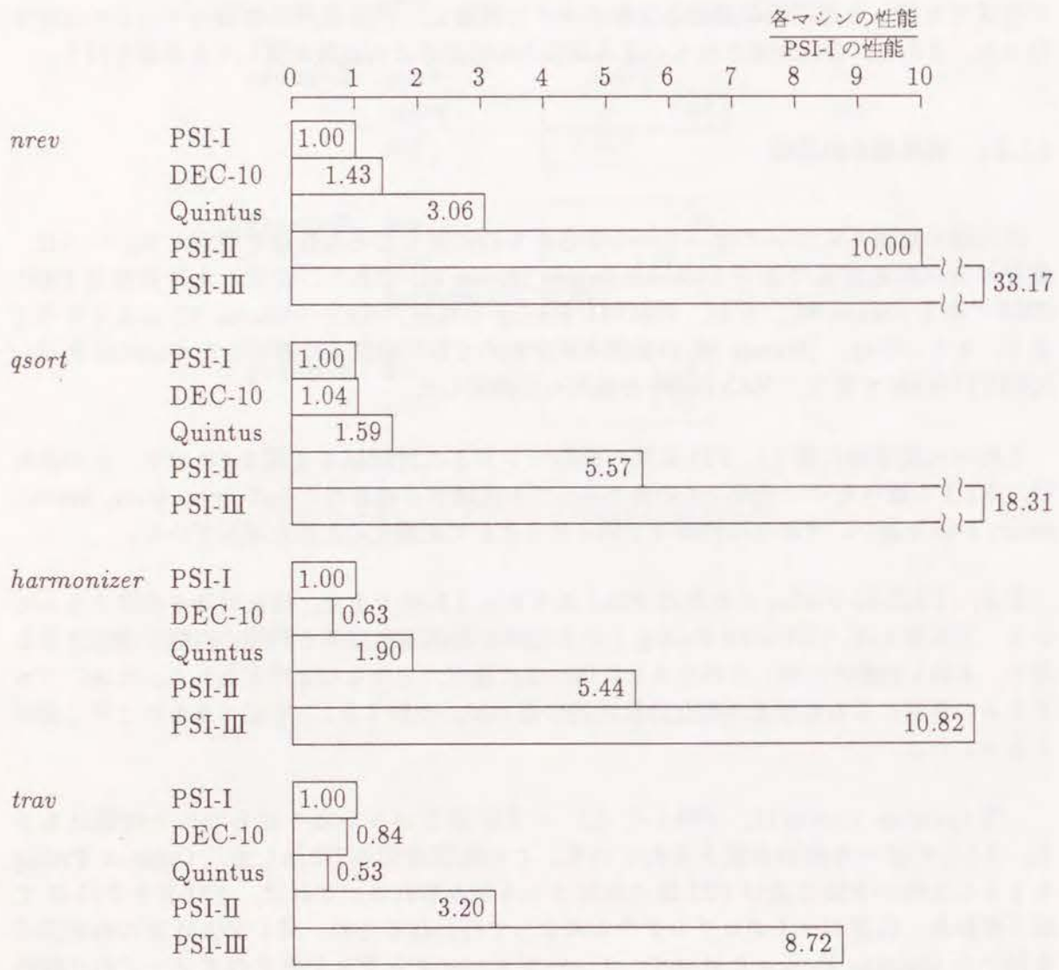


図 5-6: 汎用機との性能比較

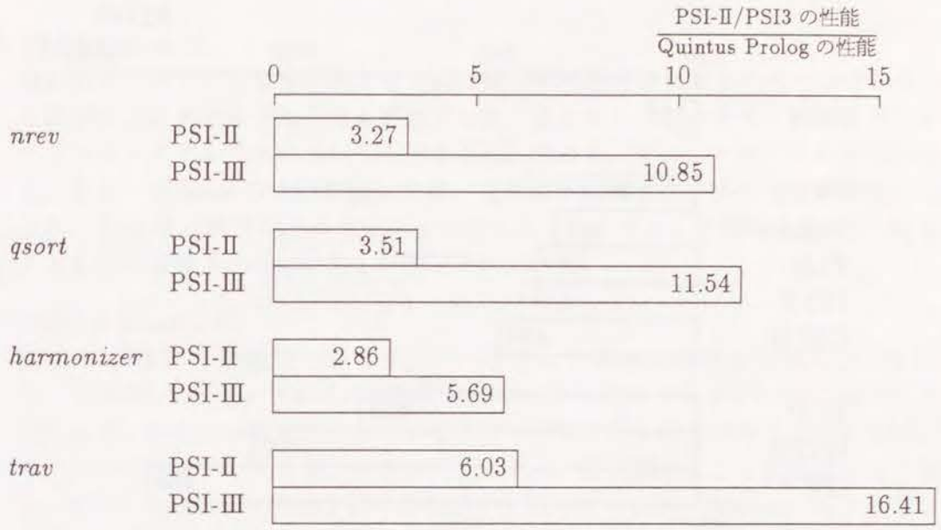


図 5-7: Quintus Prolog と PSI-II/PSI-III の性能比

5.1.2.2 専用機との比較

PSI-I の開発が成功裏に開発されたことと，その性能が高い水準を示したことなどが刺激ともなって，以後様々な論理型言語専用マシンが提案され，また実際に開発された [Yokota 87, Nakashima 90b]。それらの完成度は提案に留まったものから商用化されたものまで様々であるが，ここでは商用機またはそれに近い完成度を持ったものを対象として議論する。

まず性能を比較するためには共通の尺度が必要であるが，Drystone や Livermore Loop のような一般的なベンチマークは，論理型言語については残念ながら確立していない。また，比較的よく使われる Warren Benchmark などに関しても，文献によってその採否がまちまちである。そこで，ほとんどの文献に述べられている *append* の性能を，暫定的な尺度として用いることとした。

図 5-8は，主な論理型言語専用機と PSI 系列の推論マシンの *append* の LIPS 値である。この図から PSI-II や PSI-III の性能が高いレベルにあること，特に PSI-III の性能は最高水準にあることが明らかである。以下，各マシンについて簡単に議論する。

(1) **CHI-I** [Nakazaki 87]
PSI-I と同様，第五世代コンピュータ・プロジェクトの一環として開発された初期の専用機の一つであるが，PSI-I とは違いバックエンド・マシンであり，ESP/KL0 も

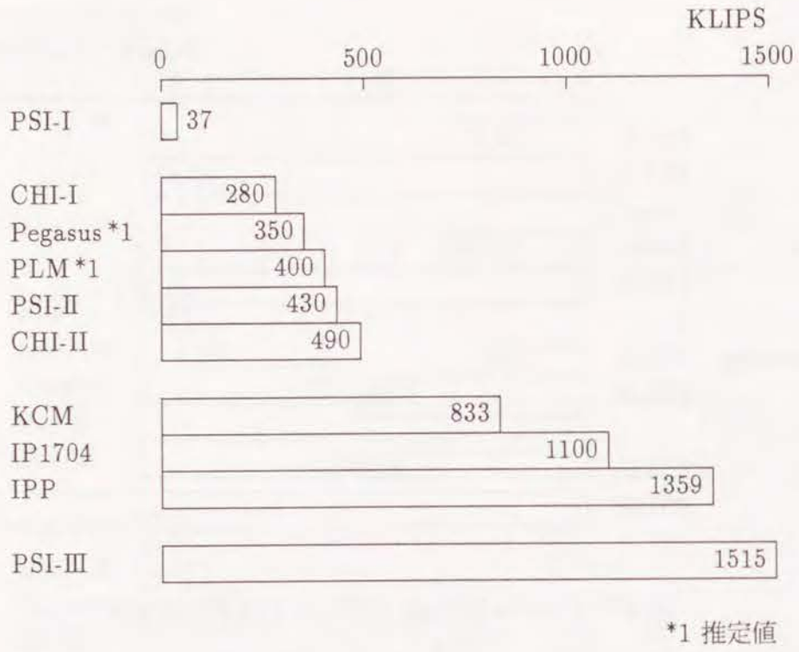


図 5-8: 論理型言語専用機の性能

サポートされていない。また，WAM をベースとする専用機としては最初に実用レベルに達したものでもある。アーキテクチャは PSI-II と同様に WAM をマイクロプログラムでエミュレートする CISC であり，簡単な命令フェッチ／デコード用のパイプラインが付加されている。また，論理素子として ECL に類似した CML (Current Mode Logic) を使用し，100 ns のサイクル・タイムを実現しているが，プリント基板 75 枚というかなり大規模な構成となっている。

(2) **Pegasus** [Seo 87, 89]
RISC [Patterson 81] の考え方を用いて実現した初めての Prolog の専用機であり，ロード，ストア，演算などの簡単な命令の他に，タグの操作を行う命令が付加されている。また，6 ステージのパイプラインの途中でデレファレンスを行う機構や，バックトラックのために引数レジスタを二組用意した「シャドー・レジスタ」などが備えられている。シャドー・レジスタは *Quick Sort* のようなプログラムに特に有効と考えられ，実際 PSI-II に対する性能比も *nrev* の 0.6 から *qsort* では 0.75 に向上している。ハードウェアは 1.5 μm の CMOS フルカスタム・チップを用いて構成されており，トランジスタ数は約 80 K とかなりコンパクトである。但し，マシン・サイクルは 200 ns と，RISC にしては少し大きめの値である。なお，*append* の性能は [Seo 87] では 239 KLIPS と報告されているが，[Seo 89] に示されているその後の改良によって

350 KLIPS 程度に改善されているものと推定される。

(3) **PLM** [Dobry 85]
WAM をベースとした専用機としては，最初期に提案されたものの一つであり，かなり詳細な設計までは少なくとも実施されたと思われる。アーキテクチャは WAM をマイクロプログラムでエミュレートする CISC であり，マシン・サイクルは 100 ns である。なお，*append* の性能に関しては，文献に不明確な点が多いため推定値とした。また，Sun などのワークステーションのコプロセッサとして動作する，PLM をベースとした商用機 Xenologic X-1 が開発されている。

(4) **CHI-II** [Habata 87]
前述の CHI-II の後継機であるが，ハードウェア構成がかなり大幅に改良されており，CMOS ゲート・アレイの使用によりマシン・サイクルが 170 ns に低下したにも関わらず，性能はかなり向上している。アーキテクチャは PSI-II と同様の CISC であり，ハードウェアの大きさ，マシン・サイクル，使用素子なども PSI-II とよく似ている。なお，*append* の性能は PSI-II を若干上回っているが，[Habata 89] に報告されている他のプログラムの性能は逆に PSI-II を下回っている [Saeki 91]。

(5) **KCM** [Benker 89]
これもマイクロプログラムによって WAM をエミュレートする CISC であるが，細かいステージングなどハードウェアの構成を工夫し，80 ns というかなり短いサイクルタイムを実現している。また，命令キャッシュとデータ・キャッシュの分離，デレファレンス用のハードウェア機構，ALU の二重化など様々な工夫がなされており，800 KLIPS を超える高い性能が達成されている。

(6) **IP1704** [Maeda 89]
Pegasus と同様に RISCy なプロセッサであるが，Pegasus や IP1704 の前身である IP704 [Kawakita 88] とは異なり，Prolog 処理のために複数サイクル要する命令を導入し，それをマイクロプログラムで処理するという CISCy な方式も取り入れている。また，シャロー・バックトラックの高速化のための，Pegasus と同様の発想に基く「ツイン・レジスタ」も備えられている。ハードウェアは 1.2 μm の CMOS VLSI チップで構成され，パイプライン・ステージは 5 段，マシン・サイクルは PSI-III と同じ 60 ns である。但し，RISCy な分だけ同じ処理に要するサイクル数が多くなるようであり，性能は PSI-III の 3/4 程度に留まっている。

(7) **IPP** [Abe 87, Kurosawa 88]
他の専用機とは大きく異なり，スーパー・ミニコンピュータに「組込まれた」エンジンである。また，論理素子として ECL を使用することによって，他の専用機を遥かに上回る 20 ns のマシン・サイクルを実現している。なお，アーキテクチャはマイクロプログラム制御による CISC であり，[Kurosawa 88] で述べられているように様々

な最適化用の命令が用いられている。*append*や他のプログラムの性能はPSI-IIIとほぼ同等であり，最高水準の性能を持つ専用機の一つである。

5.1.2.3 RISCとの比較

最近，計算機アーキテクチャの分野で最も注目を集めているのはRISCであり，これがCなどの手続型言語の分野を席捲しつつあるのは周知のとおりである。また，SOAR [Unger 84] やSPUR [Tyler 85] のように手続型以外の言語に対しても，RISCの発想を適用しようとするものも現れている。論理型言語に関しても，いくつかのRISCyなプロセッサが開発されていることは前項でも述べたが，普通のRISCで論理型言語を高速処理しようという提案もいくつか成されている。

まず実用レベルに達しているRISC上の処理系は，前述のQuintus Prologなどのように WAMをエミュレートする方式を用いており，例えばSPARC上での処理系の性能はPSI-IIIの10% ～ 20%程度である [Habata 89]。また，商用ではないものの広く利用されているSICStus Prolog [Carlsson 88] はCで書かれたWAMエミュレータを用いているため，それをMIPS [Kane 88] に実装した報告によれば性能はPSI-IIIの10%以下に留まっている [Taylor 90]。これらのエミュレーションによる方式は，コード量はPSI-IIIのようなCISCと同程度であるものの，当然のことながらWAM命令のフェッチ／デコードの手間が必要であるため*，そのオーバヘッドが性能をかなり制限してしまう。

このオーバヘッドを軽減する方法として，WAM命令をマクロとしてインライン展開する方式が考えられる。筆者がこの方式を用いて*append*をMIPSのコードに展開した結果，再帰クローズの実行サイクル数は52サイクルとなった。この値はPSI-IIIの4倍であり，現在最高速と言われている35 MHzのチップを用いても [NIKKEI 90]，PSI-IIIの1/2程度の性能に留まる。この主な原因は，分岐命令の頻度が高いことであると考えられる。

即ち，実行した命令の1/3以上である19命令が分岐命令であり，かつそのほとんど（17命令）が条件分岐命令となっている。なお，MIPSでは分岐命令は1サイクルのディレイを伴うが，ディレイド・スロットは約2/3の12命令で有効に使用され，3命令では無効，残りの4命令ではNOPとなった。一方，PSI-IIIにおいても同じ数の条件判定が必要であるが，その多くはパイプラインによるタグ判定やデレファレンスなどに吸収され，マイクロプログラムでの条件分岐は僅か3回しか行われず，この大きな違いが実行サイクルの差異に大きく影響している。

*WAMの命令列をサブルーチン・コールの列として表現する場合には，デコードの手間，即ちオペランドをサブルーチンの引数として渡す手間だけで済む。

またコード量の違いも大きく，PSI-IIIの12 wに対してMIPSでは10倍以上の132 wを必要とする*。この値は，[Seo 87] や [Borriello 87] に報告された値である5 ～ 15倍に整合しており，論理型言語の処理をRISCで行う上での重大な問題点である。またフェッチした命令のワード数も，PSI-IIIの7に対してMIPS [Korsloot 90] などの報告と整合している。これは，PSI-IIIの命令キャッシュと同じヒット率を得るためには，キャッシュ容量を7.5倍とする必要があることを意味しており，ハードウェア規模の観点から大きな問題点である。

これらの問題点，即ち条件分岐の頻発やコード量の爆発を，RISCの特徴でもある「コンパイラの努力」によって解決する方法が考えられる。即ち，述語の引数が未定義変数であるか否かをプログラマが宣言する「モード宣言」[Warren 77] の利用や，Abstract Interpretationによるモードやデータ型などの事前判定によって [Bruynooghe 87, Taylor 89]，条件分岐を除去するとともに「真に実行される」コードだけを生成する方法が提案されている。これらの中にはデレファレンスの要否までもコンパイル時に解析することにより，ほとんど全ての条件分岐を除去し，2 MLIPSという驚異的な性能を達成したとしているものもあるが [Taylor 90]，実用上はいくつかの問題点がある。

即ち，モード宣言はプログラムの「意味」には無関係のもの，例えばCのregister宣言と同様のものであると考えられるが，これを「信じて」コンパイルを行うとプログラムの意味を変えてしまう恐れがある。例えば，引数が未定義変数であるという宣言を信頼して，変数への代入を無条件に行うようなコードを生成すると，宣言が誤っていた場合には変数の値が更新されるという，論理型言語ではありえない事態が生起してしまう。

一方，Abstract Interpretationではこのような問題はないが，解析に要する時間とプログラムの修正に対する過敏な反応という二つの問題がある。即ち，[Taylor 89] には3000行程度のプログラムの解析に50分近くを要すると報告されているが，この値は実用というには程遠いものである。また，この方式は述語がどのように呼び出されているかに依存したものであるため，例えば述語pを述語qが呼び出しているとすると，（pではなく）qの修正がpのコンパイル・コードの変更を引き起こす可能性がある。従って，特に高速化が期待されるユーティリティ・ルーチンのように，様々な箇所から呼び出されている述語に対する最適なコード生成は極めて困難であり，またプログラムのモジュラリティの面からも問題が多い。更に，[Taylor 90] の値は完璧な解析が可能な極めて小規模なプログラムに関するものであり，実用規模のプログラムに関する性能は疑問視される。

以上述べたように，RISC上の論理型言語処理系の研究は注目に値するものが少なくないが，実用という観点からは当面CISCの優位は動かないものと考える。

*Choice Pointの生成，バックトラック，一般的ユニフィケーションなどのような複雑な処理はサブルーチンとしており，極端にコードを増やすようなマクロ展開は行っていない。

5.2 最適化の効果

5.2.1 ベンチマークによる評価

最適化の効果を知るためには，同じプログラムに対して最適化を施したものとそうでないものの速度差を測定する必要がある。しかし，多くの最適化手法は初めから処理系に組み込まれているため，それを除去するのは極めて困難である。そこで，簡単なベンチマーク・プログラムに対して，最適化がなされていないコードを実験的に生成し，PSI-III での速度の差を測定することとした。なお，いくつかの命令については最適化に関連する操作を修正したものを用意した。

対象としたベンチマークは，*Quick Sort* の一部である partition と，リストの n 番目の要素を求める nth の二種類である。これらのプログラム，即ち；

```
partition([],Y,L,L):-!.
partition([X|L1],Y,[X|L2],L3):- X < Y, !, partition(L1,Y,L2,L3).
partition([X|L1],Y,L2,[X|L3]):- partition(L1,Y,L2,L3).

nth(1,[E|_],E):-!.
nth(N,[_|L],E):- N1 is N - 1, nth(N1,L,E).
```

は，それぞれ図5-9，図5-10のようにコンパイルされる。また，性能の測定は以下の条件で行なった。

partition

- 第一引数は整数を要素とするリストであり，整数である第二引数よりも小さいものと大きいものとが交互に出現する。
- 第三，第四引数は未定義変数。
- 実行時間の測定はループ二回分を対象とする。即ち，図5-9に示したコードでは；

  (2) → (9) → (10) → (11) → (12) → (13) → (14) → (15) → (16) →
  (17) → (2) → (9) → (10) → (11) → (12) → (13) → (3) → (19) →
  (20) → (21) → (22) → (23) → (2)

  が対象となる。

nth

- 第一引数は正整数。
- 第二引数は任意のデータを要素とするリスト。


```
partition:
( 1)    switch_on_term  A1, partition_C1a, partition_C1,
                        partition_L1, fail
partition_L1:
( 2)    fast_try        partition_C2
( 3)    trust           partition_C3
partition_C1a:
( 4)    fast_try_me_else    partition_C2a
partition_C1:
( 5)    get_nil         A1
( 6)    get_value       A2, A3
( 7)    cut_me + proceed
partition_C2a:
( 8)    retry_me_else   partition_C3a
partition_C2:
( 9)    get_list + unify_variable + unify_variable
                        X5, X6, A1
(10)    get_list        A3
(11)    unify_value     X5
(12)    unify_variable  X7
(13)    less_than       X5, A2
(14)    cut_me
(15)    put_value       X6, A1
(16)    put_value       X7, A3
(17)    execute + switch_on_term
                        A1, partition_C1a, partition_C1,
                        partition_L1, fail
partition_C3a:
(18)    trust_me
partition_C3:
(19)    get_list + unify_variable + unify_variable
                        X5, A1, A1
(20)    get_list        A4
(21)    unify_value     X4
(22)    unify_variable  A4
(23)    execute + switch_on_term
                        A1, partition_C1a, partition_C1,
                        partition_L1, fail
```

図 5-9: 最適化された partition のコード

```
nth:
( 1)    jump_on_non_unifiable_constant
                        A1, 1, nth_C1a
nth_C1a:
( 2)    fast_try_me_else    nth_C2a
nth_C1:
( 3)    get_integer     1, A1
( 4)    get_list        A2
( 5)    unify_value     A3
( 6)    cut_me + proceed
nth_C2a:
( 7)    trust_me
nth_C2:
( 8)    get_list + unify_variable + unify_variable
                        X4, A2, A2
( 9)    subtract_constant   A1, 1, A1
(10)    execute         nth
```

図 5-10: 最適化された nth のコード

- 第三引数は未定義変数。
- 実行時間の測定はループ一回分を対象とする。即ち，図 5-10に示したコードでは；

  (1) → (8) → (9) → (10) → (1)

  が対象となる。

評価の対象とした最適化手法は，以下のものである。

(1) 組込述語の最適化
組込述語を通常述語と全く同じように取り扱うものを比較対象とすると，最適化効果が過大に見積もられるため，以下の二つを対象とした。

(a) 組込述語の引数に任意の引数レジスタを使用する最適化

(b) 一方の引数が定数である場合の算術演算組込述語の最適化

(2) バックトラックの最適化

(a) jump_on_non_unifiable_constant を用いた Clause Indexing

(b) シャロー・バックトラックの最適化（トレイルの最適化を除く）

(c) (b) におけるトレイル・バッファを用いた最適化

(d) トレイル要否判定の最適化

(3) 例外処理の最適化
4.3.2 で述べた例外が起こってからの処理を行なわないと，他の最適化ができなくなるため，例外の検出機構を評価対象とした。また，3.2.6 で述べたグレイ・ページによるメモリ割付けチェックの高速化も，例外処理の一貫として評価した。即ち，以下のものを評価対象とした。

(a) 割込を用いたフック変数の代入とトレース例外の検出。比較対象は [Carlsson 87] の方法。

(b) グレイ・ページによるメモリ割付けチェック。比較対象は 3.2.6 で述べたスタックの最大伸長量予測と，call/execute におけるそのチェック。

(4) 複合命令

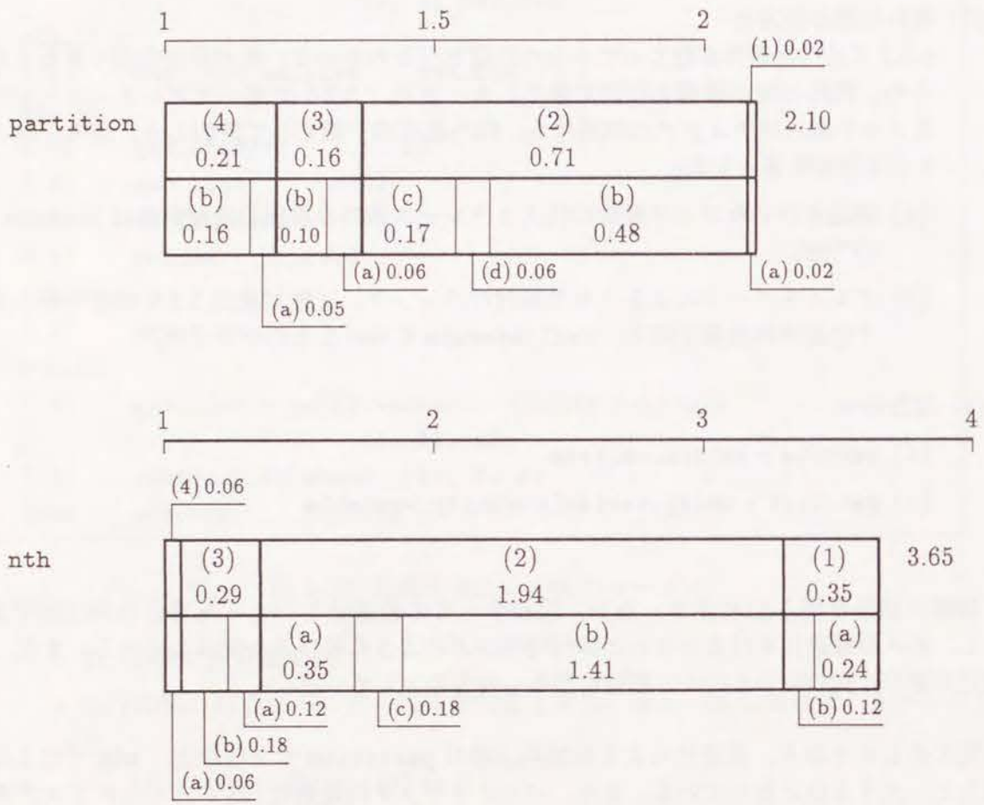
(a) execute + switch_on_term

(b) get_list + unify_variable + unify_variable

評価の結果を図 5-11に示す。なお，図はすべての最適化を行なった場合の実行時間を 1 とし，各々の最適化を行なわないと実行時間がどのように延びるかを示している。また，最適化の適用は図の右のものから順番に行なったものとしている。

図に示したとおり，最適化による性能向上率は partition で 2 倍以上，nth では 3.5 倍以上と，大きな値となっている。また，バックトラックの最適化 (2) がどちらのプログラムに対しても効果が大きく，中でもシャロー・バックトラックの最適化 (2)(b) が特に有効であることが判る。この他のバックトラックに関連のものについては，partition ではヘッド・ユニフィケーションで未定義変数の代入が行われるためトレイル・バッファ (2)(c) の効果が大きく，nth では Clause Indexing の最適化 (2)(a) により第二クローズを決定的に選択できる効果が大きい。その他のものでは，例外処理の最適化 (3) は双方に，複合命令 (4) は partition に，また組込述語の最適化 (1) は nth に対して，それぞれかなり大きな効果を発揮している。

なお，if-then-else 最適化の効果を評価するために，二つのプログラムを以下のように変更して，同じ条件での実行速度を測定した。



(1) 組込述語　(a) 引数レジスタ　(b) 定数引数

(2) バックトラック　(a) Clause Indexing　(b) シャロー・バックトラック
(c) トレイル・バッファ　(d) トレイル・チェック

(3) 例外処理　(a) 割込利用　(b) グレイ・ページ

(4) 複合命令　(a)`get_list` + `unify_variable` + `unify_variable`
(b)`execute` + `switch_on_term`

図 5-11: ベンチマークでの最適化の効果


```
partition'([],_,L,L):-!.
partition'([X|L1],Y,L2,L3):-
        (X < Y, !, L2 = [X|L21], partition'(L1,Y,L21,L3) ;
                   L3 = [X|L31], partition'(L1,Y,L2,L31)).

nth'(N,[E1|L],E2):-
        (N > 1, !, N1 is N - 1, nth'(N1,L,E2) ;
                   E1 = E2).
```

その結果，元のプログラムとの性能比は；

$$\texttt{pertition} = 1.58$$
$$\texttt{nth} = 2.13$$

と，これもかなり大きな値となることが明らかになった。

5.2.2 実用プログラムによる評価

前述のように，処理系の内部に組込まれている最適化手法を「取り外す」ことは極めて困難である。しかし，一部のものについては命令の動作を変更することなく，コンパイル・コードを変更するだけで，最適化を取り止めることが可能である。具体的には，PSI-IIのコンパイラに対して，シャロー・バックトラック及びif-then-elseの最適化の適用の可否を指令して，最適化コードとそうでないものを生成できるようにした。

これにより，大規模かつ実用的なプログラムに関する最適化効果を測定することが可能となったため，以下の三つのプログラムに関して評価を行った。

Compiler
PSI-IIのコンパイラ。ソース・プログラムは*append*とした。

Assembler
PSI-IIのマイクロプログラム・アセンブラ。ソース・プログラムは*get*系の命令を含むモジュールである。

SIMPOS
PSI-IIのオペレーティング・システムで，ウィンドウやファイルの操作を行ったもの。

上記のプログラムに関する最適化による性能向上率と，Choice Pointの統計を表5-3に示す。なおChoice Pointの統計は生成されたChoice Pointの内訳を示し，*normal*は`try(_me_else)`が，*fast*は`fast_try(_me_else)`が，それぞれ生成したものである。

表の値から，実用的なプログラムに関してもバックトラック処理の多くは最適化が可能であり，またその効果はかなり高いことが判る。なお，if-then-elseに関しては最適化の手法を熟知したプログラマが作成したものであるため（特にコンパイラ），その適用率が若干過大に評価されていると考えられる。しかし，if-then-elseが有効であるような「処理」が多いことは事実であり，プログラマのレベルに関わらずシャロー・バックトラックの最適化が有効であることは確かである。

表 5-3: バックトラックの最適化の効果

program	speed-up	Choice Points		
		normal	fast	if-then-else
Compiler	17 %	20 %	20 %	60 %
Assembler	9	56	21	23
SIMPOS	10	35	20	35

5.3 メモリ・アクセスの特性

論理型言語の処理の特徴の一つとして，メモリ・アクセスの頻度が多いことが挙げられる。即ち，実行のメカニズムがスタックのアクセスを基本としているため，アクセス回数を減らすための様々な最適化（FGO, TRO, Neck Cut Optimizationなど）を施しても，かなりの頻度でアクセスされることとなる。従って，キャッシュ・メモリの構成法などメモリ・アーキテクチャの良否が，性能に与える影響は大きいものであると考えられる。そこで，PSI-I, PSI-II, PSI-IIIの各々について，様々な角度からメモリ・アクセスの特性評価を行った。

まずPSI-Iに関しては，コマンドの種類や領域ごとのアクセス頻度，キャッシュのヒット率，キャッシュ・ミスのペナルティなどを，いくつかのプログラムを実行して計測した。また，PSI-IIを設計するための事前評価として，キャッシュの構成や容量を変化させた時の性能を，シミュレーションによって求めた。PSI-IIの評価においても，PSI-Iと同様の項目に関する測定を行い，処理方式の変更による特性の変化を調べた。また，キャッシュの変更が性能に及ぼす影響も再度シミュレーションにより求め，事前評価に基く設計の妥当性を確認した。PSI-IIIについては設計前の評価として，キャッシュの構成やTLBの構成と性能との関係を，PSI-IIの評価データに基くシミュレーションを行って計測した。

これらの評価を通じて，論理型言語の処理におけるメモリ・アクセスに際だった特徴があることや，キャッシュやTLBの構成に関する設計が妥当であることが確認された。以下，各々の推論マシンに関する評価結果について詳細に論ずる。

5.3.1 PSI-Iの評価

PSI-Iのメモリ・アクセス特性の評価は，以下の三つのプログラムの実行結果に基いて行った[Nakashima 87b]。

Window …… SIMPOSのウィンドウ・システム
BUP …… 構文解析システム
Harmonizer …… 5.1.1で述べた和音生成エキスパート

また，様々な項目に関して解析を行うことと，キャッシュの構成を変化させた時の特性の変化を調べるために，各プログラムの実行結果をトレースし，その結果を用いたシミュレーションにより評価を行った。なお，トレース・データの量は各プログラムについて約80Kサイクルである。

5.3.1.1 アクセス頻度

各プログラムを実行した時のメモリ・アクセス頻度と，そのコマンドごとの内訳を表5-4に示す。アクセス頻度は20%前後とさほど高くなく，頻繁にアクセスされるという予想が裏切られる結果となった。これは，ローカル・スタックやトレイル・スタックのバッファリングの結果と言うよりも，処理方式の問題であると考えられる。即ち，3.2.2で述べたようにインタプリタ方式には様々なオーバヘッドがあるため，メモリ・アクセスと言う真に必要な処理が行われる頻度が相対的に低くなっているものと考えられる。

コマンドの実行頻度に関しては，読出と書込の比が約3:1であることが示されている。また読出の約半数がコード（KL0の内部表現）のアクセスであると見積もられ；

$$\textit{inst-fetch}:\textit{data-read}:\textit{data-write} = 3:3:2$$
$$\Rightarrow \quad \textit{inst-fetch}:\textit{data-access} = 3:5$$

という結果となる。このコード／データ比は，コードのセマンティクス・レベルが高いことを考慮しても，データに偏った値であると言える。即ち，従来の計算機ではかなりCISCyなものでも，コード／データ比は1:1程度と言われており[Smith 82]，論理型言語処理の一つの特徴であると考えられる。従って，処理系が高速化されコード・アクセスあたりの実行時間が短縮されると，データ・アクセスの頻度は非常に高いものとなることが予想された。また，データの読出／書込の比率も[Smith 82]では2:1となっており，PSI-Iの特性はかなり書込に偏っている。論理型言語では変数への書込が二回（初期化と代入）しか起こらないにも関わらずこのような結果となったのは，変数のライフタイムが短いことを意味しているものと考えられる。なおこれらの特性は，他の評価結果[Tick 85]とも一致している。この他，スタック伸長用のwrite-stack（3.1.6参照）が書込の60～70%を占めており，コマンドの導入の妥当性が確認された。

表5-4: PSI-Iのメモリ・アクセス頻度とコマンドの実行頻度

program	read	write-stack	write	write total	total
Window	15.2%	3.5%	1.2%	4.7%	19.9%
BUP	15.6	3.5	2.2	5.7	21.3
Harmonizer	15.3	4.6	2.2	6.8	22.1

表5-5: PSI-Iのエリア別アクセス頻度

program	heap	global stack	local stack	control stack	trail stack
Window	49.6%	4.6%	16.5%	26.7%	2.6%
BUP	39.0	29.9	17.3	12.0	1.8
Harmonizer	35.2	17.7	30.3	12.8	3.8

次に，エリアごとのアクセス頻度を表5-5に示す。まず，ヒープへのアクセスの大半がコード読出によるものであるが，*Window*に関してはESPのスロットなどヒープ上のデータのアクセスが約1/4を占めている。スタックに関しては，特に*BUP*でグローバル・スタックのアクセスが多いことが目立ち，その他に関しても[Tick 85]の値を上回っている。これは，PSI-Iでは構造体共有法を用いていることが原因と考えられる。即ち，構造体共有法は頻繁に出現するリストなどの小さな構造体に関しては，複写法に比べてグローバル・スタックのアクセス回数が増加する傾向があるため，複写法の処理系を評価した[Tick 85]よりも頻度が高くなっていると思われる。また，TROに関連する「危険な」局所変数を，無条件にグローバル・スタックに割付けていることも原因の一つであろう。この他，トレイル・スタックのアクセス頻度が非常に少ないが，これはトレイル・バッファの効果と言うよりも，トレイル処理自体の頻度が低いことを示したものと考えられる。

5.3.1.2 ヒット率

各プログラムに関するキャッシュのヒット率を，表5-6に示す。なお，表にはエリアごとのヒット率も示している。どのプログラムに関してもヒット率は非常に高く，充分に満足できる値となっている。また，ローカル・スタックのヒット率が非常に高いことも明らかになったが，これはローカル・スタックのバッファリングがさほど意味を持たないこと，即ちキャッシュが実質的なバッファとなっていることを意味している。

表 5-6: PSI-I のキャッシュ・ヒット率

program	heap	global stack	local stack	control stack	trail stack	total
Window	94.1 %	92.8 %	98.9 %	99.4 %	99.6 %	96.4 %
BUP	98.2	96.8	99.0	98.2	99.7	98.0
Harmonizer	97.5	98.4	99.4	98.2	97.9	98.4

また，*Window* に関してはより詳細な評価を行い，以下のような結果を得た。

(1) write 及び write-stack のヒット率が 99 % 以上と極めて高い。write に関しては主に変数の代入に用いられるが，その場合「変数であることの確認」のために直前に必ず読出が行われるため当然の結果である。また write-stack については，スタックの伸縮が頻繁になされていることの証左である。

(2) キャッシュ・ミスのペナルティは；

$$(1 - \text{ヒット率}) \times (\text{ブロック・ロードのサイクル}) \times (\text{アクセス頻度}) = 3.6\,\% \times 6 \times 19.9\,\% = 4.3\,\%$$

で表される「最悪値」の 1/2 以下である 2.1 % であった。これは，3.1.6 で述べた同期方式の工夫や，Load Through 方式の採用の効果である。

(3) Store Back 方式を Store Through に変更すると約 5 % の性能低下となる。また，キャッシュ・ミスのうちストア・バックを伴うものと伴わないものの比率が約 50 : 1

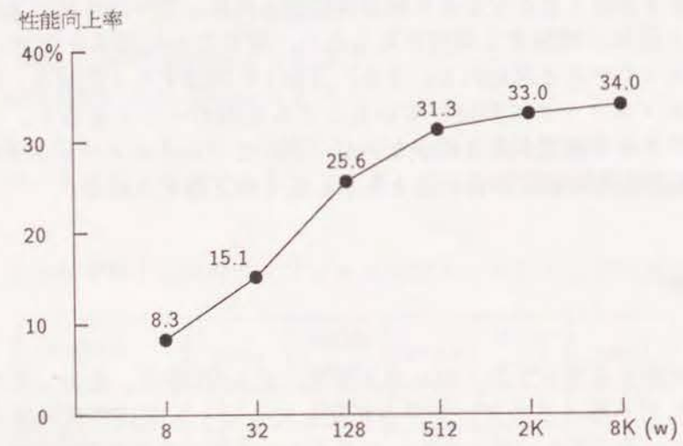


図 5-12: キャッシュ容量と性能の関係 (PSI-I)

であること，ストア・バックされるキャッシュ・ライン (4w) の平均更新回数は 11.4 回であることなど，Store Back 方式の有効性を裏付ける結果も得られた。

(4) キャッシュの容量を変化させた時の性能をシミュレートしたところ，図 5-12に示す結果となった。この結果と All in Cache の性能が 1.36 であることから，キャッシュ容量の削減が可能ではないかと考えた。そこで，他のプログラムも含めて，4Kw のダイレクト・マップ方式の性能を測定した結果，いずれについても性能低下は 3 % 以下であることが明かとなった。

5.3.2 PSI-II の評価

3.2.6 で述べたように PSI-II のキャッシュ・メモリは，実装規模の削減のためにその容量を PSI-I の 1/2 である 4Kw とし，また構成方式もセット・アソシアティブからダイレクト・マッピングに変更した。この変更は 5.3.1 で述べた評価，即ち容量削減による性能低下は 3% 以下であるという結果に基くものであった。

しかし，PSI-II では ESP/KL0 の処理方式を大幅に変更したため，PSI-I での評価結果がそのまま適用しうるか否かについて，若干の疑問が残っていた。即ち；

(1) 処理方式の変更によって主記憶アクセス特性が変化しないか。

(2) 性能向上に伴う主記憶アクセス頻度の相対的増加により，キャッシュ・ミスヒットの影響が大きくならないか。

(3) ダイレクト・マッピングへの変更によって，スラッシングが頻発しないか。

などの点が問題となっていた。

そこで，PSI-I で行ったものと同様の評価を再度実施し，これらの疑問に対する回答を得ることとした。なお，評価は PSI-I と同様に，プログラムの実行結果のトレースと，それを用いたシミュレーションにより行った。また，実行したプログラムは PSI-II のコンパイラであり，トレース・データ量は約 160K サイクルである。

5.3.2.1 アクセス頻度

図 5-13は，キャッシュ・メモリのアクセス頻度を示したものである。また，コマンドの実行頻度とエリア別のアクセス頻度も併せて示している。キャッシュ・メモリのアクセス頻度は 60% を越えており，PSI-I と比べると約 3 倍に増加している。この結果は，PSI-I と PSI-II ではメモリ・アクセス回数に大差はなく，性能が 3 倍以上に向上した分だけ相対的に頻度が増加したことを示している。

コマンドの実行頻度に関しては，命令の読出，データの読出，及びデータの書込の比率がほぼ 1:1:1 であり，PSI-I よりも若干データ書込の比率が多くなっているが，概ね同じような傾向である。従って，高頻度のデータ・アクセスという論理型言語の特徴が再度確認された。また，write-stack が書込全体に占める割合が，PSI-I よりも若干増加して約 3/4 となっている。これは，3.2.6 で述べた write-stack の改良によるものであると考えられる。

エリアごとのアクセス頻度については，ローカル・スタックへのアクセス頻度が全体の約半分，データ・エリアの 3/4 を占めているのが特徴である。この値は [Tick 85] とは一致し

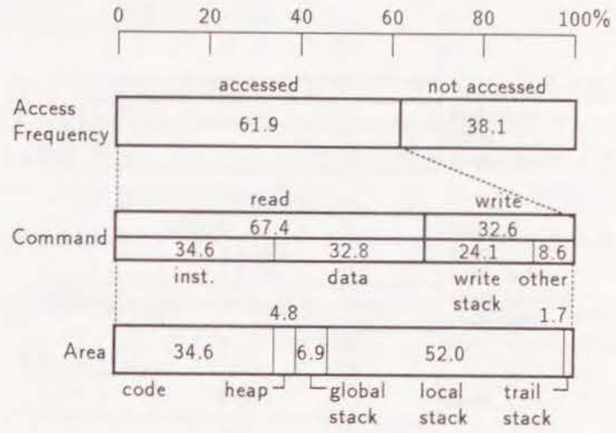


図 5-13: PSI-II のメモリ・アクセス頻度

ているが，PSI-I と比較するとグローバル・スタックのアクセス頻度が減少している。これは前述のように，PSI-I が構造体共有法を用いているのに対し，PSI-II や [Tick 85] では構造体複写法を用いているためである。

5.3.2.2 ヒット率

図 5-14にコマンド／エリアごとのヒット率を示す。命令に比べてデータ・アクセスのヒット率が高く，特に書込でのヒット率が PSI-I と同様に極めて高い。エリア別のヒット率については，ローカル・スタックのヒット率が非常に高く，実質的にローカル・スタック・バッファの役割を果たしていることが判る。全体のヒット率は 96.8% と充分に高率であり，心配されたヒット率の低下は重大なものではなかったと考えられる。またミスヒットの内，ストア・バックを伴うものは極めて少なく，Store Back 方式の有効性が再確認された。

次に，キャッシュ容量半減の影響を調べるために，同じデータを用いて 4Kw × 2set のキャッシュ・メモリのヒット率と性能をシミュレートした。図 5-15は PSI-II のキャッシュ・メモリと，4Kw×2set のキャッシュ・メモリを比較したものである。なお，性能についてはヒット率が 100% の時の性能を 1 として正規化してある。ヒット率，性能とも差異は極めて小さく，容量削減が誤った選択ではなかったことが確認された。

なお，キャッシュ・ミス・ペナルティの最悪値，即ち；

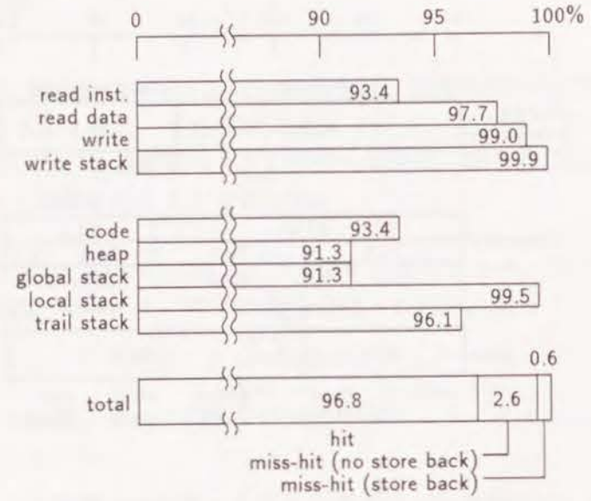


図 5-14: PSI-II のキャッシュ・ヒット率

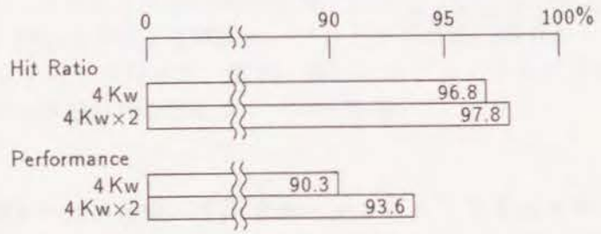


図 5-15: キャッシュ容量と性能の関係 (PSI-II)

(1－ヒット率)×(ブロック・ロードのサイクル)×(アクセス頻度)
＝3.2％×6×61.9％＝11.9％

と，実際のペナルティ 9.7％ の差は，PSI-I と比べてかなり小さくなっている。これは，アクセス頻度が増加したために，キャッシュの動作完了を CPU が待合せる確率が高くなったためであると考えられる。

5.3.3 PSI-III の評価

3.3.5 で述べたように，PSI-III では命令／データ・キャッシュの分離と，アドレス変換バッファ TLB の導入という，メモリ・アーキテクチャに関する二つの大きな変更を行った。まずキャッシュに関しては，それぞれの容量をどのようなものにするかが，VLSI チップに搭載可能なトランジスタ数との関係で問題となった。また，TLB に関しては，ミスが発生した時のペナルティが大きいため，そのヒット率を充分に高くする必要があった。

そこで，PSI-II の評価に用いたデータに基き，キャッシュ容量，TLB の容量や構成方式を変化させたシミュレーションによりヒット率を測定し，設計の参考データとして用いることとした。

5.3.3.1 キャッシュ容量

キャッシュ容量の決定に際しては，VLSI チップに搭載可能な容量で充分な性能が得られるか否かが最大の問題であった。そこで，キャッシュの容量を変化させてヒット率を測定したところ，図 5-16に示す結果が得られた。即ち，命令キャッシュは 1Kw，データ・キャッシュは 2Kw 程度までは，容量の対数にほぼ比例してヒット率が向上している。

一方，チップに搭載可能な容量は，命令／データ・キャッシュを合わせて 1Kw 内外と見積もられた。従って；

(a) 命令キャッシュ＝512w，データ・キャッシュ＝512w
(b) 命令キャッシュ＝256w，データ・キャッシュ＝1Kw

が考えられた。しかし，(a) の場合はデータ・キャッシュのヒット率が，(b) の場合は命令キャッシュのヒット率がかなり低くなる。

さて，キャッシュ・ミス・ペナルティの大きな要因となるブロック・ロードに要するサイクル数は，主記憶を構成する DRAM の速度がほとんど向上していないため，単純にマシン・サイクルに反比例して増加し，PSI-II の 2.5 ～ 3 倍となることが予想された。更に，もう一つの要因であるアクセス頻度が増加することにより，最悪値の増加だけではなく，実際のペナルティとの差もより小さくなると考えられる。従って，ヒット率の低下による性能への悪影響が相対的に拡大することから，これらの構成では充分な性能を得ることが難しいと判断した。

そこで，データ・キャッシュのデータ・アレイをチップ外部に置くことを検討した結果，アクセス時間 15ns の高速 RAM を用いれば，チップ内部に搭載した場合と同等のマシン・

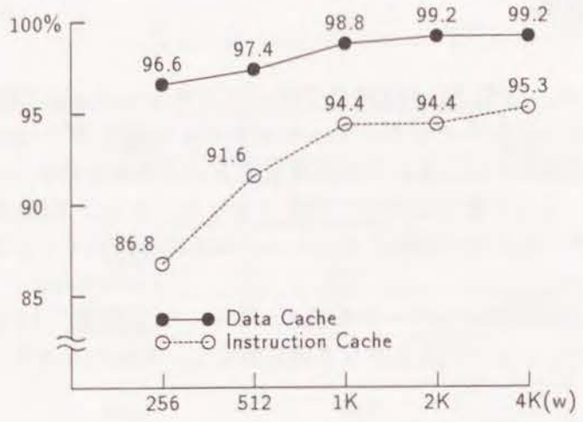


図 5-16: 命令／データ・キャッシュのヒット率

サイクルが得られることが明らかになった。その結果，命令キャッシュは1Kw，データ・キャッシュは4Kwの容量とすることができ，充分に高いヒット率を実現することが可能となった。

5.3.3.2 TLB

TLBのヒット率に関しては，ミスヒット時のペナルティがキャッシュ・メモリに比べてかなり大きいと予想されるため，極めて高い値であることが要求される。ことにPSI-IIIではハードウェア設計の簡単化のために，TLBミスの際の処理をマイクロプログラムで行うこととしたため，ヒット率には細心の注意が必要となった。そこで，TLBに関しては，容量だけではなく構成方式に関しても，いくつかのバリエーションを考えて評価を行った。

まず，命令用TLBについては，ダイレクト・マッピング方式と2セットのセット・アソシアティブ方式の双方について，16 ～ 64エントリの場合のヒット率を測定した。その結果，図5-17に示すようにダイレクト・マッピングでは64エントリ，セット・アソシアティブでは32エントリで100%のヒット率が得られた。

なお，命令アドレスの分布を詳しく調べると，使用したページの内の約1/3はプログラムの本体から離れた領域にあった。このような分布を示したのは，システムで用意しているライブラリを共有コード方式で実現しているためである。この方式では本体とライブラリが連続領域を形成しないため，命令アドレスの分布が離散的になる傾向がある。従って，TLBの容量が，ライブラリを含めたプログラム・サイズを満たすものであっても，TLBミスが発生する可能性がある。そこで命令用アドレス変換バッファの構成は，評価結果からかなり

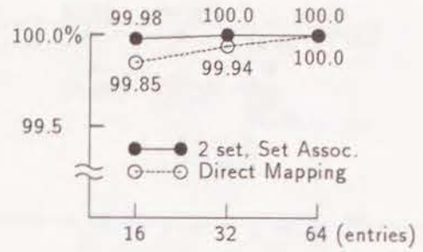


図 5-17: 命令用TLBのヒット率

の余裕があると見られる，64エントリのセット・アソシアティブ方式とすることとした。

一方，データのアドレス分布については，複数のエリアがアクセスされるため，離散傾向が更に強まることが容易に予測できる。また，スタックの伸びを押さえる処理方式をとっているため，特にローカル・スタックとトレイル・スタックについてはスタック・ボトム付近へのアクセスが集中すると予想される。即ち，使用するアドレスの分布は；

$$\text{trail-stack} \approx \text{local-stack} < \text{global-stack} \ll \text{heap}$$

となることが予想される。

図5-18は，2セットのセット・アソシアティブ方式のアドレス変換バッファのヒット率を，容量とエントリ・アドレスの生成方法を変化させて測定した結果である。この中でNaiveは，ページ番号の下位ビットをそのままエントリ・アドレスとしたものである。この方法は上記の性質を完全に無視しているため，スタック間でページの奪いあいが頻発し，非常に低いヒット率となっている。特にトレイル・スタックについては容量によらず58.8%という惨澹たる結果になっている。

次にArea Oriented-Iは，エリア番号とページ番号の下位ビットを結合したものをエントリ・アドレスとする方法であり，スタック間でのページの奪いあいがなくなるためにかなり高いヒット率が得られる。しかし，この方法はエリア間で異なるアドレス分布を無視したものであるため，離散傾向が強いヒープに関してのヒット率が悪くなってしまう。実際，ヒープのヒット率は96.9 ～ 96.2%であり，充分な値であるとは言えない。

そこで，エントリ・アドレスEAを下式によって生成する方法Area Oriented-IIを評価した。

$$EA\langle n:n-2\rangle = page\#\langle n:n-2\rangle \oplus area\#$$
$$EA\langle n-3:0\rangle = page\#\langle n-3:n\rangle \qquad (\oplus:\text{Exclusive-OR})$$

この方法では，スタック間でのページの奪いあいが抑止されるとともに，ヒープのために多くのエントリが確保される。従って，図5-18に示すように高いヒット率が得られ，64

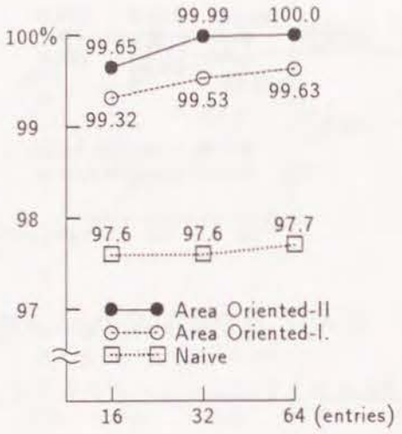


図 5-18: データ用 TLB のヒット率

エントリの場合には 100% となった。そこで，アドレス変換バッファの構成を，64 エントリ，2 セットのセット・アソシアテイブ方式とし，エントリ・アドレスの生成方式は Area Oriented-II の方法を採用することとした。但し，プロセス切換時の TLB ミスを緩和するために，プロセスの識別番号やエリアがプロセス固有か共有かを加味し，3.5.5 で述べたエントリ・アドレス生成方式；

$$EA\langle 4:2\rangle = page\#\langle 4:2\rangle \oplus area\#$$

$$EA\langle 1:0\rangle = \begin{cases} page\#\langle 1:0\rangle, & \text{for heap} \\ page\#\langle 1:0\rangle \oplus pid\langle 1:0\rangle, & \text{for stacks} \end{cases}$$

を採用した。

第6章
結論

冒頭でも述べたように本研究の最大の目的は，推論マシンのアーキテクチャはいかにあるべきかを検討し，高速かつ実用的なプロセッサを実現することであった。また，新たな最適化手法の考案と実証や，並列推論マシンの実現も，本研究の遂行に当っての大きな目的であった。これらの目的については本論文で明らかにしてきたように，全て高い水準で達成することができ，以下に示すような様々な成果を挙げることができた。

(1) 論理型言語向きの基本アーキテクチャの提案：
アーキテクチャの研究に関しては，まず処理対象である論理型言語に対する深い考察に基き，タグ操作や記憶管理の実現方式が性能に密接に関連していることを明らかにした。即ち第2章で述べたように，論理型言語における基本操作であるユニフィケーションでは，データ型の判定とデレファレンスという，タグの抽出と判定操作が中核となっていることを明かにした。またこのタグ判定操作が，LISP 処理のような二方向分岐中心のものとは異り，多方向分岐，それも分岐の方向に偏りが少ないものであることを見い出した。更に，構造体のユニフィケーションと，論理型言語のもう一つの特徴であるバックトラックの実現のためには，独立に伸縮する複数のスタックが必要であることも示した。

このような考察に基づき，第3章の冒頭で述べたアーキテクチャの三大基本方針，即ち；

- 高速ユニフィケーションのためのタグ操作機構
- 複数スタックの効率的実現のための記憶管理機構
- ハードウェアによる並行処理を柔軟に制御する水平型マイクロプログラム

を提案したことが第一の成果である。

(2) 基本アーキテクチャの有効性実証：
第3章で論じた三つの逐次型推論マシン PSI-I, PSI-II 及び PSI-III の設計を，上記の基本アーキテクチャに基づいて行い，その有効性をそれぞれの推論マシンが持つ高い性能により証明したことが第二の成果である。特に PSI-I の設計時点では推論マシンのアーキテクチャは未知の領域であったため，基本アーキテクチャの確立と実証は極めて意義深いものであった。

また個々の要素に関しても，3.1 節で述べたテーブルを用いたタグ多方向分岐機構や，スタックを独立した記憶空間である「エリア」に対応させるアーキテクチャな

ど，従来にないハードウェアの構成方式を提案することができたことも大きな成果であった。更に，3.1節に示したハードウェアの利用状況の評価や，5.3節に示したメモリ・アクセス特性の評価は，後継機であるPSI-II, PSI-IIIの重要な役割を果たしたに留まらず，推論マシン・アーキテクチャ全般に大きな影響を与えた。

(3) インタプリタ方式とコンパイル方式の評価：
3.2節及び5.1節で述べたように，PSI-IにWAMをベースとしたコンパイル方式の処理系を試験実装し，PSI-I本来の処理方式であるインタプリタ方式との比較評価を行った。この研究を通じてコンパイル方式がインタプリタ方式の2～3倍の性能であることと，処理に必要な情報を適切な時点で提示しているか否かが性能差の主要因であることを明かにしたことが第三の成果である。即ち，当時提案段階にあったWAMの有効性を先駆的に証明したことは極めて意義深く，その後の推論マシン・アーキテクチャの研究に大きく貢献するものであった。

(4) コンパイル方式向きアーキテクチャの提案：
3.2節で述べた第二の逐次型推論マシンPSI-IIは，上記の研究成果に基づいて設計され，WAMをベースとした推論マシンとしての先駆的役割を果たした。この設計を通じ，論理型言語のコンパイル方式による実行に関する様々な新しいアイデア，即ち；

- 組込述語のための命令アーキテクチャ
- 処理モードを加味した命令デコード機構
- グレイ・ページによるメモリ割当検出機構
- 割込を用いた例外検出機構

などを提案／実装したことが，第四の成果である。また，これらの機構やPSI-Iの評価結果に基づくハードウェア／マイクロ命令アーキテクチャの改良により，同じコンパイル方式を用いたPSI-I上の実験処理系の1.5～4倍，インタプリタ方式に比べると3～11倍という，大幅な性能向上を達成できたことも大きな成果であった。更に，430 KLIPS (*append*) という当時では世界最高の性能をPSI-IIが達成したことは，推論マシンの研究に大きなインパクトを与えた。

(5) 論理型言語向きパイプライン方式の提案：
3.3節で述べた第三の逐次型推論マシンPSI-IIIでは，論理型言語処理に向いたパイプライン方式という画期的なアーキテクチャを提案した。即ち，従来の命令フェッチ／デコード，オペランドのアドレス計算／フェッチに加え，データ型の判定とデレファレンスという特有の機能をパイプライン化した構成方式を考案したことが第五の成果である。この方式は，PSI-I以来の基本方針であるハードウェアによるタグ操作の並行処理の概念を時間軸方向に拡大したものであるとともに，条件分岐のハードウェア化というパイプライン・マシンの高速化についての一つの解答でもあった。このパイプライン方式の導入により，PSI-IIIの性能はPSI-IIに比べて更に2～3.5倍に向上し，1.5 MLIPS (*append*) という現時点での世界最高水準の性能を達成することができた。

(6) 最適化手法の提案と評価：
本研究の開始時点においては，論理型言語に関する最適化手法としては，2.4節で述べたTail Recursion OptimizationとClause Indexingが知られているのみであり，研究すべき事項が数多く存在する未熟な分野であった。従って，第4章で述べた最適化手法，即ち；

- 組込述語の引数受渡し方式
- 引数が変数であるクローズに関するIndexing
- Neck Cutに関するShallow BacktrackとTidy Trailの最適化
- タグを用いたトレイル要否判定
- コンパイル時に予見できない例外事象の処理方式

の新たな考案を，第六の成果として挙げることができる。また5.2節で述べたように，これらの手法はいずれもPSI-II, PSI-IIIに適用されて高い効果を発揮したとともに，その多くは汎用機上の処理系にも適用できることから，論理型言語処理の高速化という普遍的な課題に対する貢献も成しえたものと考える。

(7) 並列推論マシンの実現：
3.4節で述べた通り，PSI-Iを要素プロセッサとするMulti-PSI/v1，64個のPSI-IIからなるMulti-PSI/v2，及び256個のPSI-IIIからなるPIM/mの三つの並列推論マシンを実現したことが第七の成果である。また，ストリーム通信の最適化，構造データの要素の更新，プロセッサ内外での実時間ガベージ・コレクションなど，これらの並列推論マシンの開発を通じて得られた知見により，並列論理型言語の研究を大きく発展させることができた。

以上述べたような成果を達成しつつ，本研究は成功裏に一応の区切りを迎えることができたが，推論マシンに関する研究には多くの課題が残されているものと考える。本論文を結ぶに当たって，これらの課題に関して簡単に触れることとする。

第一の課題は，言うまでもなくPSI-IIIの改良である。PSI-IIIでは初めてパイプライン方式を導入したため，インタロックや分岐に関する制御は論理型言語の一般的特徴に基づく簡単なものとした。問題はこの「一般的特徴」の仮定が，実際のプログラムに対してどの程度成立するかであり，これを確認するための評価が必要である。現在，基本的な評価作業を進めている段階であるが[Saeki 91]，パイプライン・フラッシュのペナルティが予想以上に大きいプログラムもあり，詳細な解析とそれに基づく改良方式の検討を行う予定である。更に，最近注目されているSuper ScalerやVLIWの考え方を，論理型言語向きのパイプラインに採り入れることも検討すべきである。

第二の課題は，処理方式の研究，特に最適化方式を更に追求することである。5.1節でも述べたように，Abstract Interpretationを用いた解析により，大域的な最適化を行う研究が盛んに行われている。現時点では実用性の面で問題が多いが，解析時間やモジュラリティに関する欠点を解決するために，言語仕様を含めて実用化に向けた研究が必要であろう。ま

た，誤っていても意味を変えないようにモード宣言を利用する方法や，モード宣言が正誤を解析する手法の研究も，実用的システムと言う観点からは重要な項目であると考える。

第三の課題は，並列推論マシンの要素プロセッサのアーキテクチャ研究である。これまでに筆者らが行った研究は，ネットワークの構成やプロセッサ間通信の方式など，「並列」に重点をおいたものであった。逆に，並列マシンの性能の基本となる要素プロセッサ自体の高速化については，PSI-IIIにおける実時間ガベージ・コレクションのサポート機構を除いては，深い研究が成されていなかったとも言える。一方，Multi-PSI/v2上で様々な応用プログラムが開発されるに従って，並列論理型言語特有のコンテクスト・スイッチが予想以上に高頻度であるとともに，この処理の「重さ」が性能にかなりの悪影響を及ぼしていることが指摘されている。従って，「軽い」コンテクスト・スイッチを実現する機構，例えばRegister WindowやMulti-Thread Pipelineなどの導入も含めて，要素プロセッサのアーキテクチャを再検討する必要があろう。

本研究の過程を振り返ると，10年に満たない期間の間に三つの推論マシンを開発し，しかもその性能を数十倍に向上するという，飛躍的進歩を成しえたことに大きな喜びを感ずる。この進歩を継続することは必ずしも容易ではないだろうが，本研究で得た様々な知見に基づき上記の課題を着実に解決することで，推論マシン，論理型言語，ひいてはAIを始めとする高度な計算機応用を発展させるために，一層の貢献を成しえると信ずるものである。

謝辞

本研究をまとめるに際し，御指導，御鞭撻，ならびに格別の御配慮を賜った京都大学・富田眞治教授に，深甚の謝意を表する。また本論文の執筆に当たって懇切な御指導を頂いた，京都大学・堂下修司教授，柴山潔助教授に，深く感謝の意を表する。

本研究は，第五世代コンピュータ・プロジェクトの一貫として，(財)新世代コンピュータ技術開発機構(ICOT)の御協力と御指導のもとで，三菱電機(株)情報電子研究所において行なわれた。

ICOTでは，同研究所所長・淵一博氏，同所研究部長・内田俊一氏に，研究の開始当初から暖かい御指導と御援助を頂いた。ここに深く感謝の意を表する。また，研究の遂行に当たり直接御指導，御助言を頂いた，同所第一研究室長・瀧和男氏，同所第二研究室長・近山隆氏，日本電気(株)C&C研究所・横田実氏の三氏に，筆者の衷心よりの謝意を捧げる。特に，PSI-Iのアーキテクチャ設計は瀧，横田両氏の御尽力に負うところが大きく，また近山氏はESP, KL0, KL1を設計されたのみならず，これらの言語の処理方式の検討にも多大な貢献をされた。更にICOTに在籍されていた山本明氏（沖電気工業(株)），西川宏氏（松下技研(株)），後藤厚宏氏（日本電信電話(株)），木村康則氏（富士通研究所(株)），宮崎敏彦氏（沖電気工業(株)），六沢一昭氏（同），現在在籍されている市吉伸行氏（ICOT第七研究室長代理），上田和紀氏（同研究部），稲村雄氏（同第一研究室），及び沖電気工業(株)の三井正樹氏，吉田裕之氏の諸兄には，さまざまな局面で多大な御協力を頂いた。ここに深く感謝する次第である。

また三菱電機(株)において御指導，御鞭撻を頂いた，大阪大学・首藤勝教授（元情報電子研究所副所長），情報電子研究所所長・曽我正和氏，同所副所長・武藤達也氏，情報家電開発センター長・田中千代治氏，情報電子研究所次世代計算機開発部長・岩瀬正氏，同所ビジネスコンピュータ開発部・上田尚純氏，コンピュータ製作所・沓沢啓二氏に，深く感謝の意を表する。特に，上田氏には研究の全過程を通じて，暖かい御援助を頂いたことを感謝したい。また，研究の遂行に当たって御指導，御協力を頂いた，鷹取東朋氏（伊丹製作所），中屋雅夫氏（LSI研究所），山田通裕氏（北伊丹製作所），益田嘉直氏（コンピュータ製作所），京敬人氏（同），池田守弘氏（同），田辺隆司氏（同），三石彰純氏（同），大上貴英氏(Mitubishi Electric Research Laboratories)，中島克人氏（情報電子研究所）の諸兄に深く感謝する。特に，各推論マシンのアーキテクチャ設計に際しての，中島氏の多大な御尽力に感謝したい。また，ハードウェアの開発では武田保孝氏（情報電子研究所），ファームウェアの開発では立野裕和氏（同），コンパイラの開発では近藤誠一氏（現ICOT第二研究室）の絶大なる御協力を頂いた。ここに深く感謝する。

更に，推論マシンの開発には極めて多くの方々に御尽力頂いた。特にハードウェア／ファームウェアの開発に御尽力された灘敏明氏（コンピュータ製作所），深沢雄氏（同），

岩山洋明氏（同），佐伯稔氏（同），宮内信仁氏（同），安藤秀樹氏（LSI研究所），町田浩之氏（同），中西知嘉子氏（同），古谷清広氏（同），安田憲一氏（同），村澤靖氏（情報電子研究所），小森隆三氏（同），山崎高日子氏（同），高橋勝己氏（同），宮崎進一氏（三球電気(株)），小笠原兼幸氏（大井電子(株)），坂俣勝則氏（同），千葉信行氏（同），田村洋氏をはじめとする三菱電機エンジニアリング(株)の諸兄，杉森一三氏をはじめとするアイテック阪神(株)の諸兄，またここにお名前をあげることのできなかった，ソフトウェア開発などに御尽力頂いた全ての方々に，深い感謝の意を表したい。

最後に，研究を行なう間，筆者を常にバックアップしてくれた我が家族に，心から感謝する。

発表論文一覧

[Nakashima 84] 中島　浩，田辺　隆司，瀧　和男：逐次型推論マシン ψ のハードウェア — コンソール・プロセサの構成と機能 —，情報処理学会第 29 回全国大会，1B-9, pp. 59–60 (1984).

[Nakashima 85] 中島　浩，三石　彰純，瀧　和男：PSI の性能評価 (2)，情報処理学会第 30 回全国大会，1C-3, pp. 155–156 (1985).

[Nakashima 86] 中島　浩，近山　隆，中島　克人：マルチ PSI 要素プロセッサ PSI-II の最適化手法，情報処理学会第 33 回全国大会，7B-4, pp. 151–152 (1986).

[Nakashima 87a] 中島　浩，中島　克人，瀧　和男：PSI へのコンパイラ向き Prolog 命令の試験実装と評価，情報処理学会論文誌, Vol. 28, No. 10, pp. 1091–1095, 1987.

[Nakashima 87b] 中島　浩，瀧　和男，中島　克人，三石　彰純：パーソナル逐次型推論マシン PSI の評価 — 実行速度とハードウェア各部の性能について —，情報処理学会論文誌, Vol. 28, No. 12, pp. 1298–1305, 1987.

[Nakashima 87c] H. Nakashima and K. Nakajima. Hardware Architecture of the Sequential Inference Machine: PSI-II. *Proc. 4th IEEE Symposium on Logic Programming*, pp. 104–113, 1987.

[Nakashima 87d] 中島　浩，立野　裕和，木村　康則：PSI-II における KL0 実行順序系組込述語の実現方式，情報処理学会第 35 回全国大会，6B-6, pp. 675–676 (1987).

[Nakashima 90a] 中島 浩，武田　保孝：PSI-II のメモリ・アーキテクチャ評価，情報処理学会計算機アーキテクチャ研究会，No. 80-8, pp. 57–64 (1990).

[Nakashima 90b] 中島　浩：専用 VLSI プロセッサの具体例 — VLSI 記号処理プロセッサ —，情報処理, Vol. 32, No. 4, pp. 486–491 (1990).

[Nakashima 90c] 中島　浩，武田　保孝，中島　克人：PIM/m 要素プロセッサのアーキテクチャ，並列処理シンポジウム JSPP'90, pp. 145–151 (1990).

[Nakashima 90d] H. Nakashima, Y. Takeda, K. Nakajima, H. Andou and K. Furutani. A Pipelined Microprocessor for Logic Programming Languages. *Proc. 1990 Intl. Conf. on Computer Design*, pp. 355–359, 1990.

[Nakashima 90e] 中島　浩，武田　保孝，瀧　和男：PIM/m 要素プロセッサのアーキテクチャ，情報処理学会第 40 回全国大会，2L-6, pp. 1183–1184 (1990).

[Taki 84] K. Taki, M. Yokota, A. Yamamoto, H. Nishikawa, S. Uchida, H. Nakashima and A. Mitsuishi. Hardware Design and Implementation of the Personal Sequential Inference Machine (PSI). *Proc. Intl. Conf. on Fifth Generation Computer Systems 1984*, pp. 398–409, 1984.

[Nakajima 86a] K. Nakajima, H. Nakashima, M. Yokota, K. Taki, S. Uchida, H. Nishikawa, A. Yamamoto and M. Mitsui. Evaluation of PSI Micro-Interpreter. *Proc. Compcon Spring 86*, pp. 173–177, 1986.

[Yoshida-H 86] 吉田　裕之，池田　守宏，中島　浩，横田　実，中島　克人：マルチPSI要素プロセッサPSI-IIのメモリ管理とプロセス管理，情報処理学会第33回全国大会，7B-5, pp. 153–154 (1986).

[Nakajima 87] 中島　克人，稲村　雄，中島　浩，近藤　誠一，吉田　裕之：PSI-IIの性能評価(1) — 概要と速度評価 —，情報処理学会第35回全国大会，6B-7, pp. 677–678 (1987).

[Yoshida-H 87] 吉田　裕之，中島　浩，中島　克人：PSI-IIの性能評価(3) — マイクロ命令動的評価 —，情報処理学会第35回全国大会，6B-8, pp. 679–680 (1987).

[Taki 87] K. Taki, K. Nakajima, H. Nakashima and M. Ikeda. Performance and Architectural Evaluation of the PSI Machine. *Proc. 2nd Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 1987.

[Rokusawa 88] K. Rokusawa, N. Ichiyoshi, T. Chikayama and H. Nakashima. An Efficient Termination Detection and Abortion Algorithm for Distributed Processing Systems. *Proc. 1990 Intl. Conf. on Parallel Processing*, Vol. 1, pp. 18–22, 1988.

[Takeda 88] Y. Takeda, H. Nakashima, K. Masuda, T. Chikayama and K. Taki. A Load Balancing Mechanism for Large Scale Multiprocessor Systems and Its Implementation. *Proc. Intl. Conf. on Fifth Generation Computer Systems 1988*, pp. 978–986, 1988.

[Tateno 89] 立野　裕和，近藤　誠一，中島　浩，中島　克人：PSI-IIの機械命令セット評価，情報処理学会計算機アーキテクチャ研究会，No. 74-3, pp. 1–8 (1989).

[Saeki 91] 佐伯　稔，中島　浩，立野　裕和，池田　守宏，田嶋　隆二：PIM/mフロントエンド・プロセッサの速度性能評価，情報処理学会第42回全国大会 (1991).

[Machida 91] H. Machida, H. Andou, C. Ikenaga, H. Nakashima, A. Maeda, M. Nakaya. A 1.5 MLIPS 40-bit AI Processor. *Proc. Custom Integrated Circuits Conf.*, 1991.

参考文献

[Abe 87] S. Abe, T. Bandoh, S. Yamaguchi, K. Kurosawa and K. Kiriyama. High Performance Integrated Prolog Processor IPP. *Proc. 14th Intl. Symp. on Computer Architecture*, pp. 100–107, 1987.

[AMD 85] Am29300 Family Handbook. Advanced Micro Devices, Inc., 1985.

[Benker 89] H. Benker, J. M. Beacco, M. Dorochevsky, Th. Jeffré, A. Pöhlmann, N.Noyé and B. Poterie. KCM: A Knowledge Crunching Machine. *Proc. 16th Intl. Symp. on Computer Architecture*, 1989.

[Borriello 87] G. Borriello, A. R. Cherenson, P. B. Danzig and M. N. Nelson. RISCs or CISCs for Prolog: A Case Study. *Proc. 2nd Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 1987.

[Bowen 81] D. L. Bowen. DEC System-10 Prolog User's Manual. *Dept. of Artificial Intelligence, Univ. of Edinburgh*, 1981.

[Bruynooghe 87] M. Bruynooghe, G. Janssens, A. Callebaut, B. Demoen. Abstract Interpretation: Towards the Global Optimisation of Prolog Programs. *Proc. 4th IEEE Symposium on Logic Programming*, pp. 192–204, 1987.

[Carlsson 86] M. Carlsson. On Compiling Indexing and Cut for the WAM. *Research Report 86011, Swedish Institute of Computer Science*, 1986.

[Carlsson 87] M. Carlsson. Freeze, Indexing, and Other Implementation in the WAM. *Proc. 4th Intl. Conf. on Logic Programming*, pp. 40–58, 1987.

[Carlsson 88] M. Carlsson and J. Widen. SICStus Prolog Users Manual, *SICS Research Report R88007B*, 1988.

[Carlsson 89] M. Carlsson. On the Efficiency of Optimizing Shallow Backtracking in Compiling Prolog. *Proc. 6th Intl. Conf. on Logic Programming*, 1989.

[Colmerauer 82] A. Colmerauer. Prolog II: Manuel de Réfénce et Modèle Thèorique. *Groupe Intelligence Artificielle, Université Aix-Marseille II*, 1982.

[Colwell 85] R. P. Colwell, C. Y. Hitchcock III, E. D. Jensen, H. M. B. Sprunt and C. P. Koller. Instruction Sets and Beyond: Computers, Complexity, and Controversy. *IEEE Computer*, Vol. 18, No. 9, pp. 8-19, 1985.

[Chikayama 83a] T. Chikayama. ESP — Extended Self-Contained Prolog — as a Preliminary Kernel Language of Fifth Generation Computers. *New Generation Computing*, Vol. 1, No. 1, Ohmu-Sha, 1983.

[Chikayama 83b] T. Chikayama, M. Yokota and T. Hattori. Fifth Generation Kernel Language. *Proc. Logic Programming Conf. '83*, 1983.

[Chikayama 84] T. Chikayama. Unique Features of ESP. *Proc. Intl. Conf. on Fifth Generation Computer Systems 1984*, pp. 292–298, 1984.

[Chikayama 87] T. Chikayama and Y. Kimura. Multiple Reference Management in Flat GHC. *Proc. 4th Intl. Conf. on Logic Programming*, pp. 276–293, 1987.

[Chikayama 88] T. Chikayama, H. Sato and T. Miyazaki. Overview of the Parallel Inference Machine Operating System (PIMOS). *Proc. Intl. Conf. on Fifth Generation Computer Systems 1988*, pp. 230–251, 1988.

[Clark 86] K. Clark and Steve Gregory. PARLOG: Parallel Programming in Logic. . *ACM Trans. on Programming Languages and Systems*, Vol. 8, No. 1, pp. 1–49, 1986.

[Debray 86] S. K. Debray and D. S. Warren. Detection and Optimization of Functional Computation in Prolog. *Proc. 3rd Intl. Conf. on Logic Programming*, 1986.

[Dobry 85] T. P. Dobry, A. M. Despain and Y. N. Patt. Performance Studies of a Prolog Machine Architecture. *Proc. 12th Intl. Symp. on Computer Architecture*, pp. 180–190, 1985.

[Eriksson 84] L-H. Eriksson and M. Rayner. Incorporating Mutable Arrays into Logic Programming. *Proc. 2nd Intl. Conf. on Logic Programming*, 1984.

[FAIRCHILD 82] FAST Data Book. Fairchild Inc., 1982.

[Furukawa 84] 古川　康一：Prolog総論，情報処理, Vol. 25, No. 12, pp. 1313–1318 (1984).

[Goto 88a] A. Goto, Y. Kimura, T. Nakagawa and T. Chikayama. Lazy Reference Counting: An Incremental Garbage Collection Method for Parallel Inference Machines. *Proc. 5th Intl. Conf. and Symp. on Logic Programming*, pp. 1240–1256, 1988.

[Goto 88b] A. Goto, M. Sato, K. Nakajima, K. Taki and A. Matsumoto. Overview of the Parallel Inference Machine Architecture (PIM). *Proc. Intl. Conf. on Fifth Generation Computer Systems 1988*, pp. 208–229, 1988.

[Goto 91] 後藤　厚宏：並列型推論マシンのアーキテクチャ，情報処理，Vol. 32, No. 4, pp. 458–467 (1991).

[Habata 87] S. Habata, R. Nakazaki, A. Konagaya, A. Atarashi and M. Uemura. Co-Operative High Performance Sequential Inference Machine: CHI. *Proc. 1987 Intl. Conf. on Computer Design*, 1987.

[Habata 89] 幅田　伸一，小長谷　明彦，新　淳，横田　実：逐次型推論マシンCHI-IIの性能評価，*TM 734, ICOT*, 1989.

[Hagiwara 77] 萩原　宏：マイクロプログラミング，産業図書 (1977).

[Hirano 90] 平野　善芳，後藤　厚宏：並列論理型言語KL1のコンパイル方式の改良，並列処理シンポジウムJSPP'90, pp. 281–288 (1990).

[Ichiyoshi 87] N. Ichiyoshi, T. Miyazaki and K. Taki. A Distributed Implementation of Flat GHC on the Multi-PSI. *Proc. 4th Intl. Conf. on Logic Programming*, pp. 257–275, 1987.

[Ichiyoshi 88] N. Ichiyoshi, K. Rokusawa, K. Nakajima and Y. Inamura. A New External Reference Management and Distributed Unification for KL1. *Proc. Intl. Conf. on Fifth Generation Computer Systems 1988*, pp. 904–913, 1988.

[Ichiyoshi 91] 市吉　伸行：論理型言語指向の並列処理方式，情報処理，Vol. 32, No. 4, pp. 435–449 (1991).

[ICOT 90] Proc. Workshop on Concurrent Programming and Parallel Processing. ICOT, 1990.

[Ida 85] 井田　哲雄：LISPマシンのアーキテクチャ，情報処理，Vol. 26, No. 7, pp. 732–740 (1985).

[Ikeda 87] 池田　守宏，村澤　靖，吉田　裕之，近山　隆：PSI-IIにおけるESPサポート用ファームウェア，情報処理学会第35回全国大会，6B-5, pp. 673–674 (1987).

[Inamura 87] 稲村　雄，近藤　誠一，中島　克人：PSI-IIの性能評価 (2) — 機械命令評価 —，情報処理学会第35回全国大会，6B-8, pp. 679–680 (1987).

[Inamura 89] Y. Inamura, N. Ichiyoshi, K. Rokusawa and K. Nakajima. Optimization Technique Using the MRB and Their Evaluation on the Multi-PSI/V2. *Proc. North American Conf. on Logic Programming 1989*, pp. 907–921, 1989.

[Kane 88] G. Kane. MIPS RISC Architecture. Prentice-Hall, 1988.

[Kaneda 91] 金田　悠紀夫，松田　秀雄：逐次型推論マシンのアーキテクチャ，情報処理，Vol. 32, No. 4, pp. 450–457 (1991).

[Kawakita 88] S. Kawakita, M. Saito, Y. Hoshino, Y. Bandai and Y. Kobayashi. An Integrated AI Environment for Industrial Expert Systems. *Proc. Intl. Workshop on AI for Industrial Applications 1988*, 1988.

[Kimura 87] Y. Kimura and T. Chikayama. An Abstract KL1 Machine and Its Instruction Set. *Proc. 4th IEEE Symposium on Logic Programming*, pp. 468–477, 1987.

[Kimura 89] 木村　康則，西崎　慎一郎，中越　靖行，平野　善芳：KL1 のクローズインデキシング方式，並列処理シンポジウム JSPP'89, pp. 187–194 (1989).

[Kimura 90] 木村　康則，近山　隆：並列論理型言語 KL1 の多重参照管理によるガベージコレクション，情報処理学会論文誌, Vol. 31, No. 2, pp. 316–327, 1990.

[Kondoh 88] S. Kondoh and T. Chikayama. Macro Processing in Prolog. *Proc. 5th Intl. Conf. and Symp. on Logic Programming*, pp. 466–480, 1988.

[Korsloot 90] M. Korlsoot and H. M. Mulder. Sequential architecture models for Prolog: A Performance Comparison. *Proc. 7th Intl. Conf. and Symp. on Logic Programming*, pp. 49–66, 1990.

[Kowalski 74] R. Kowalski. Predicate Logic as Programming Language. *Information Processing 74*, pp. 569–574, 1974.

[Kurosawa 88] K. Kurosawa, S. Yamaguchi, S. Abe and T. Bandoh. Instruction Architecture for a High Performance Integrated Prolog Processor IPP. *Proc. 5th Intl. Conf. and Symp. on Logic Programming*, pp. 1506–1530, 1988.

[Maeda 89] K. Maeda, et al. Mechanisms for Achieving Parallel Operations in a Sequential VLSI AI Processor. *Proc. 3rd Annual Parallel Processing Symp.*, 1989.

[Masuda 88] K. Masuda, H. Ishizuka, H. Iwayama, K. Taki and E. Sugino. Preliminary Evaluation of the Connection Network for the Multi-PSI system. *Proc. 8th European Conf. on Artificial Intelligence*, pp. 18–23, 1988.

[Machida 91] H. Machida, H. Andou, C. Ikenaga, H. Nakashima, A. Maeda, M. Nakaya. A 1.5 MLIPS 40-bit AI Processor. *Proc. Custom Integrated Circuits Conf.*, 1991.

[Miyauchi 87] 宮内　信仁，木村　康則，近山　隆，久門　耕一：MRB による多重参照管理方式 —KL1 処理系における一括型 GC の特性評価 —. 情報処理学会第 35 回全国大会，2Q-7, pp. 711–712 (1987).

[Miyazaki 88] 宮崎　敏彦：並列論理型言語 KL1 の実現方式と並列 OS の記述，電子情報通信学会論文誌，Vol. J71-D, No. 8, pp. 1423-1432 (1988).

[Morris 79] F. L. Morris. A Time-and Space-Efficient Garbage Compaction Algorithm. *Communication of the ACM*, Vol. 20, No. 10, 1979.

[Nakajima 86a] K. Nakajima, H. Nakashima, M. Yokota, K. Taki, S. Uchida, H. Nishikawa, A. Yamamoto and M. Mitsui. Evaluation of PSI Micro-Interpreter. *Proc. Compcon Spring 86*, pp. 173–177, 1986.

[Nakajima 86b] 中島　克人，横田　実，山本　明：PSI の性能評価 (3)，情報処理学会第 31 回全国大会，3C-2, pp. 55–56 (1986).

[Nakajima 87] 中島　克人，稲村　雄，中島　浩，近藤　誠一，吉田　裕之：PSI-II の性能評価 (1) — 概要と速度評価 —，情報処理学会第 35 回全国大会，6B-7, pp. 677–678 (1987).

[Nakajima 89] K. Nakajima, Y. Inamura, N. Ichiyoshi, K. Rokusawa and T. Chikayama. Distributed Implementation of KL1 on the Multi-PSI/V2. *Proc. 6th Intl. Conf. on Logic Programming*, 1989.

[Nakajima 90a] 中島　克人，稲村　雄，市吉　伸行：疎結合並列マシン Multi-PSI 上での KL1 分散処理系におけるプロセッサ間通信の評価，並列処理シンポジウム JSPP'90, pp. 369–376 (1990).

[Nakajima 90b] K. Nakajima and N. Ichiyoshi. Evaluation of Inter-Processor Communication in the KL1 Implementation on the Multi-PSI. *Proc. 1990 Intl. Conf. on Parallel Processing*, Vol. 1, pp. 613–614, 1990.

[Nakashima 84] 中島　浩，田辺　隆司，瀧　和男：逐次型推論マシン ψ のハードウェア —コンソール・プロセサの構成と機能 —，情報処理学会第 29 回全国大会，1B-9, pp. 59–60 (1984).

[Nakashima 85] 中島　浩，三石　彰純，瀧　和男：PSI の性能評価 (2)，情報処理学会第 30 回全国大会，1C-3, pp. 155–156 (1985).

[Nakashima 86] 中島　浩，近山　隆，中島　克人：マルチ PSI 要素プロセッサ PSI-II の最適化手法，情報処理学会第 33 回全国大会，7B-4, pp. 151–152 (1986).

[Nakashima 87a] 中島　浩，中島　克人，瀧　和男：PSI へのコンパイラ向き Prolog 命令の試験実装と評価，情報処理学会論文誌, Vol. 28, No. 10, pp. 1091–1095, 1987.

[Nakashima 87b] 中島　浩，瀧　和男，中島　克人，三石　彰純：パーソナル逐次型推論マシン PSI の評価 —実行速度とハードウェア各部の性能について —，情報処理学会論文誌, Vol. 28, No. 12, pp. 1298–1305, 1987.

[Nakashima 87c] H. Nakashima and K. Nakajima. Hardware Architecture of the Sequential Inference Machine: PSI-II. *Proc. 4th IEEE Symposium on Logic Programming*, pp. 104–113, 1987.

[Nakashima 87d] 中島　浩，立野　裕和，木村　康則：PSI-IIにおけるKL0実行順序系組込述語の実現方式，情報処理学会第35回全国大会，6B-6, pp. 675–676 (1987).

[Nakashima 90a] 中島 浩，武田　保孝：PSI-IIのメモリ・アーキテクチャ評価，情報処理学会計算機アーキテクチャ研究会，No. 80-8, pp. 57–64 (1990).

[Nakashima 90b] 中島　浩：専用VLSIプロセッサの具体例 — VLSI記号処理プロセッサ—，情報処理, Vol. 32, No. 4, pp. 486–491 (1990).

[Nakashima 90c] 中島　浩，武田　保孝，中島　克人：PIM/m要素プロセッサのアーキテクチャ，並列処理シンポジウムJSPP'90, pp. 145–151 (1990).

[Nakashima 90d] H. Nakashima, Y. Takeda, K. Nakajima, H. Andou and K. Furutani. A Pipelined Microprocessor for Logic Programming Languages. *Proc. 1990 Intl. Conf. on Computer Design*, pp. 355–359, 1990.

[Nakazaki 87] R. Nakazaki, A. Konagaya, S. Habata, H. Simazu, M. Uemura and M. Yamamoto. Design of a High-Speed Prolog Machine (HPM). *Proc. 14th Intl. Symp. on Computer Architecture*, pp. 191–197, 1987.

[NIKKEI 90] 日経エレクトロニクス：オープン・システム時代を快走するワークステーション，日経エレクトロニクス，No. 516, pp. 93–122 (1990).

[Okuno 84] 奥野　博：第3回Lispコンテストと第1回Prologコンテストの課題案，情報処理学会記号処理研究会，No. 28-4 (1984).

[Okuno 85] H. Okuno. The Report of the Third Lisp Contest and the First Prolog Contest. 情報処理学会記号処理研究会，No. 33-4 (1985).

[Patterson 81] D. A. Patterson and C. H. Séquin. RISC I: Reduced Instruction Set VLSI Computer. *Proc. of 8th Intl. Symp. on Computer Architecture*, 1981.

[Pohm 83] A. V. Pohm and O. P. Agrawal. High-Speed Memory Systems. Reston Publishing Company, Inc., 1983.

[QUINTUS 85] Quintus Prolog Reference Manual. Quintus Computer Systems, Inc., 1985.

[Rokusawa 88] K. Rokusawa, N. Ichiyoshi, T. Chikayama and H. Nakashima. An Efficient Termination Detection and Abortion Algorithm for Distributed Processing Systems. *Proc. 1990 Intl. Conf. on Parallel Processing*, Vol. 1, pp. 18–22, 1988.

[Saeki 91] 佐伯　稔，中島　浩，立野　裕和，池田　守宏，田嶋　隆二：PIM/mフロントエンド・プロセッサの速度性能評価，情報処理学会第42回全国大会，2H-5, pp. 6-25–26 (1991).

[Seo 87] K. Seo and T. Yokota. Pegasus: A Risc Processor for High-Performance Execution of Prolog Programs. *Proc. of Intl. Conf. on Very Large Scale Integration*, pp. 261-274, 1987.

[Seo 89] K. Seo and T. Yokota. Design and Fabrication of Pegasus Prolog Processor. *Proc. Intl. Conf. on Very Large Scale Integration*, 1989.

[Shapiro 86] E. Shapiro. Concurrent Prolog: A Progress Report. *IEEE Computer*, Vol. 19, No. 8, pp. 44-58, 1986.

[Smith 82] A. J. Smith. Cache Memories. *Computing Surveys*, Vol. 14, No. 3, pp.473–530, 1982.

[Takagi 85] 高木　茂行，近山　隆，横田　実，瀧　和男：if-then-elseの導入によるESP実行効率の向上，情報処理学会第31回全国大会，3C-1, pp. 53–54 (1985).

[Takeda 88] Y. Takeda, H. Nakashima, K. Masuda, T. Chikayama and K. Taki. A Load Balancing Mechanism for Large Scale Multiprocessor Systems and Its Implementation. *Proc. Intl. Conf. on Fifth Generation Computer Systems 1988*, pp. 978–986, 1988.

[Taki 84] K. Taki, M. Yokota, A. Yamamoto, H. Nishikawa, S. Uchida, H. Nakashima and A. Mitsuishi. Hardware Design and Implementation of the Personal Sequential Inference Machine (PSI). *Proc. Intl. Conf. on Fifth Generation Computer Systems 1984*, pp. 398–409, 1984.

[Taki 87] K. Taki, K. Nakajima, H. Nakashima and M. Ikeda. Performance and Architectural Evaluation of the PSI Machine. *Proc. 2nd Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 1987.

[Tanaka 91] 田中　英彦：論理型言語指向の推論マシンの位置付けと開発の現状，情報処理，Vol. 32, No. 4, pp. 415–420 (1991).

[Tateno 87] 立野　裕和，池田　守宏，西川　宏：PSI-IIのGC (1) — 多重論理空間のGC —，情報処理学会第35回全国大会，6B-3, pp. 669–670 (1987).

[Tateno 89] 立野　裕和，近藤　誠一，中島　浩，中島　克人：PSI-IIの機械命令セット評価，情報処理学会計算機アーキテクチャ研究会，No. 74-3, pp. 1–8 (1989).

[Taylor 89] A. Taylor. Removal of Dereferencing and Trailing in Prolog Compilation. *Proc. 6th Intl. Conf. and Symp. on Logic Programming*, 1989.

[Taylor 90] A. Taylor. LIPS on MIPS: Results from a Prolog Compiler for a RISC. *Proc. 7th Intl. Conf. and Symp. on Logic Programming*, pp. 174–185, 1990.

[Tick 85] E. Tick. Prolog Memory-Reference Behavior. *Technical Report No.85-281, Computer Systems Lab., Stanford Univ.*, 1985.

[Tomita 88] 富田　眞治，村上　和彰：計算機システム工学，昭晃堂 (1988).

[Touati 87] H. Touati and A. Despain. An Empirical Study of the Warren Abstract Machine. *Proc. 4th IEEE Symposium on Logic Programming*, pp. 114–124, 1987.

[Tylor 85] G. S. Tylor, P. N. Hilfinger, J. R. Larus, D. A. Patterson and B. G. Zorn. Evaluation of the SPUR Lisp Architecture. *Proc. 12th Intl. Symp. on Computer Architecture*, pp. 444-452, 1985.

[Uchida 88] S. Uchida, K. Taki, K. Nakajima, A. Goto and T. Chikayama. Research and Development of the Parallel Inference System in the Intermediate Stage of the FGCS Project. *Proc. Intl. Conf. on Fifth Generation Computer Systems 1988*, pp. 16–36, 1988.

[Ueda 85] K. Ueda. Guarded Horn Clauses. *TR 103, ICOT*, 1985. (Also in *Concurrent Prolog: Collected Papers, The MIT Press*, 1987.)

[Ueda 86a] K. Ueda. Guarded Horn Clauses: A Parallel Logic Programming Language with the Concept of a Guard. *TR 208, ICOT*, 1986. (Also in *Programming of Future Generation Computer, North-Holland*, 1987.)

[Unger 84] D. Unger, R. Blau, P. Foley, D. Samples and D. Patterson. Architecture of SOAR: Smalltalk on a RISC. *Proc. 11th Intl. Symp. on Computer Architecture*, pp. 188–197, 1984.

[Warren 77] D. H. D. Warren. Implementing Prolog — Compiling Predicate Logic Programs. *Research Report No. 39, 40, Dept. of Artificial Intelligence, Univ. of Edinburgh*, 1977.

[Warren 80] D. H. D. Warren. An Improved Prolog Implementation which Optimises Tail Recursion. *Proc. Logic Programming Workshop*, 1980.

[Warren 83] D. H. D. Warren. An Abstract Prolog Instruction Set. *Technical Report 309, Artificial Intelligence Center, SRI International*, 1983.

[Yasui 82] 安井　裕：LISP マシン，情報処理，Vol. 23, No. 8, pp. 757–772 (1982).

[Yokoi 84] T. Yokoi and S. Uchida. Sequential Inference Machine: SIM — Its Programming and Operating System. *Proc. Intl. Conf. on Fifth Generation Computer Systems 1984*, pp. 70–81, 1984.

[Yokota 84] M. Yokota, A. Yamamoto, K. Taki, H. Nishikawa, S. Uchida, K. Nakajima and M. Mitsui. A Microprogrammed Interpreter for the Personal Sequential Inference Machine. *Proc. Intl. Conf. on Fifth Generation Computer Systems 1984*, pp. 410–418, 1984.

[Yokota 85] 横田　実，近山　隆，西川　宏，中島　克人：PSI におけるオブジェクトサポート，情報処理学会第 30 回全国大会，1C-1, pp. 153–154 (1985).

[Yokota 87] 横田　実：逐次型 Prolog マシン，人工知能学会誌，Vol. 2, No. 4 (1987).

[Yokota 91] 横田　実：論理型言語の逐次実行処理方式：情報処理，Vol. 32, No. 4, pp. 421–434 (1991).

[Yoshida-H 86] 吉田　裕之，池田　守宏，中島　浩，横田　実，中島　克人：マルチ PSI 要素プロセッサ PSI-II のメモリ管理とプロセス管理，情報処理学会第 33 回全国大会，7B-5, pp. 153–154 (1986).

[Yoshida-H 87] 吉田　裕之，中島　浩，中島　克人：PSI-II の性能評価 (3) — マイクロ命令動的評価 —，情報処理学会第 35 回全国大会，6B-8, pp. 679–680 (1987).

[Yoshida-K 88] K. Yoshida and T. Chikayama. A'UM — A Stream-Based Concurrent Object-Oriented Language —. *Proc. Intl. Conf. on Fifth Generation Computer Systems 1988*, pp. 638–649, 1988.

[Yoshida-T 90] T. Yoshida, M. Matsuo, T. Ueda and Y. Saito. A Strategy for Avoiding Pipeline Interlock Delays in a Microprocessor. *Proc. 1990 Intl. Conf. on Computer Design*, pp. 355–359, 1990.

付録

付録 A　ESP/KL0

A.1　オブジェクト指向機能

A.1.1　基本的な言語仕様

ESP プログラムは，クラスというモジュールを単位として記述される。クラスは状態変数であるスロット，他のクラスとのインタフェースとなる特殊な述語であるメソッド，及び外部には公開しないローカル述語から構成される。

例えば，ファイル・システムを実現するクラスである file_system は，以下のように記述される。

```
class file_system has
        :open(Class, File_Spec, Obj):-
                :new(Class, Obj),
                open_file(Obj, File_Spec) ;
instance
component
        file_status ;

        :read(Obj, Data):-
                read_from_disk(Obj!file_status, Data);
        :write(Obj, Data):-
                write_to_disk(Obj!file_status, Data);
        :close(Obj):-
                close_file(Obj);
local
        open_file(Obj, File_Spec):- ...
        read_from_disk(Status, Data):- ...
        write_to_disk(Status, Data):- ...
        close_file(Obj):- ...
end.
```

上記の例で，:open はクラス・メソッドと呼ばれるメソッドであり，:new によってクラス file_system に属する「オブジェクト」を生成する。個々のオブジェクトは互いに独立し

た状態変数(スロット)を持つ一方，それを操作するメソッドを共有している。即ち，ファイルの状態を保持するスロットである file_status は個々のオブジェクト（即ちファイル）に固有のものであるが，ファイルを操作するメソッドである :read, :write, :close は file_system に属する全てのオブジェクトに共通のものである。

またスロット file_status は，外部に対して秘匿されている。即ち，外部とのインタフェースであるメソッドの引数は Obj は，オブジェクト自身を保持する変数であり，それを経由したスロットへのアクセス，例えば Obj!file_spec は同じクラスの中でのみ許される*。従って，スロットが保持するデータの構造，意味などを，プログラムの他の部分とは全く独立に設計／変更することができる。同様に，ローカル述語 open_file, read_from_disk, write_to_disk, close_file も，外部から直接呼出すことはできないため，実現方法の詳細を隠蔽することができる。

なお，スロット参照操作 Obj!file_status はマクロであり，例えば；

```
read_from_disk(Obj!file_status, Data)
```

は，スロットを参照する KL0 の組込述語 get_slot を用いて；

```
get_slot(Obj, file_status, X), read_from_disk(X, Data)
```

と展開される。このように，ゴール引数として現れたマクロを別のゴールに展開する機能は，ESP のマクロの重要な特徴である。また，スロットを更新する操作もマクロとして用意されており；

```
Obj!Slot := Data
```

は，組込述語 set_slot を用いて；

```
set_slot(Obj, Slot, Data)
```

と展開される。

さて，メソッドの第一引数 Obj は，単にスロットへのアクセス・パスを表現しているだけではなく，オブジェクトがどのクラスに属しているかの情報ももっている。即ち，メソッド :read の呼出し；

```
..., :read(Obj, Data), ...
```

において，Obj がクラス file_system に属していれば，上記のメソッドが呼出される。しかし，Obj が別のクラス，例えば terminal_io に属していれば，terminal_io の中で定義された :read が呼出される。このように，オブジェクトによってメソッドの意味が定まるという性質は，オブジェクト指向言語の重要な特徴である。これにより「対象によって異なる類似した操作」を一つの共通した名前で呼出すことができ，例えば OS とユーザ・プログラムのインタフェースの統一化などに極めて有効である。

*ESP にはこのように外部から見えないスロットである Component Slot の他，外部に公開するスロットである Attribute Slot がある。

A.1.2 継承

ESP のオブジェクト指向機能で重要な概念の一つが「継承」である。ESP のクラスは他のクラスのメソッドを継承し，あたかも自身の中に他のクラスで定義されたメソッドが存在するかのように見せることができる*。

例えば前述のファイル・システムに，行単位の read/write 操作を付加することを考える。このためのクラス file_system_with_line は；

```
class file_system_with_line has
nature file_system ;

instance
        :read_line(Obj, Line):- ...
        :write_line(Obj, Line):- ...
            ⋮
end.
```

のように，クラス file_system を継承し，追加機能だけを定義することによって実現される。即ち，クラス file_system が持つ open/close などの基本機能を，そのまま file_system_with_line の機能とすることができる。

また，:read_line や :write_line の定義が端末との入出力にも適用できる場合には；

```
class with_line has
instance
        :read_line(Obj, Line):- ...
        :write_line(Obj, Line):- ...
            ⋮
end.
```

のようにクラス with_line を定義し，クラス file_system_with_line は；

```
class file_system_with_line has
nature file_system, with_line;
end.
```

のように，二つのクラスを継承するだけで定義することも可能である。

この他，他のクラスのメソッドに対して，一定の制約条件を付加するような継承も可能である。例えば，一つのファイルに対して複数の open を禁じる場合；

*スロットも継承される。

```
class exclusive_open_file_system has
nature file_system ;
before :open(Class, File_Spec):- not_opened(File_Spec);
                ⋮
end.
```

のように定義すると，:open は；

```
:open(Class, File_Spec):- not_opend(File_Spec),
        :new(Class, Object), open_file(Object, File_Spec);
```

のように実行される。この before が付されたメソッドは *Before Demon* と呼ばれ，同じ名前のメソッドを実行するための前提条件や前処理のために用いることができる。また，after が付された *After Demon* もあって，メソッドの実行結果のチェックなどの後処理に用いられる*。

ESP が持つこれらの高度な継承機能，即ち複数のクラスの継承や，Before/After Demon は，ソフトウェアの「部品化」を促し，ソフトウェアの生産性向上に大きく貢献している。

A.1.3 オブジェクトとクラスの実現方式

前述のように，ESP におけるオブジェクト指向の概念は，プログラマにとって極めて有用であり，特に高度なモジュール化によるソフトウェア生産性の向上に多大な効果をもたらしている。しかしながら，言語の抽象度は Prolog に比べてさらに高度なものとなっており，処理系に対する負担も増加している。

処理系にとって最も負担となるのは，メソッド呼出しの処理である。ESP のメソッドが具体的に何を行うかは，その第一引数であるオブジェクトがどのクラスに属しているかによって定まる。即ち，実行時にオブジェクトが定まるまでは，メソッドに対応するコードを決定することができない。そこで，オブジェクトとメソッド名による，コードの探索を実行時に行わなければならない。

また，スロットのアクセスについても，クラスの外部に公開する Attribute Slot が存在することと，スロットが継承されることから，名前以外の情報（例えばオブジェクト内でのスロットの相対位置）をコンパイル時に定めることができない。従って，オブジェクトとスロット名によるスロットの物理的位置の決定も実行時に行われる。

コードやスロットの探索は，図 A-1に示すデータ構造を用いて行われる [Yokota 85, Ikeda 87]。まず，コンパイル時にクラス内で定義されたメソッド / スロットの名前と，コード・

*Demon 以外のメソッドは *Principal Predicate* と呼ばれる。

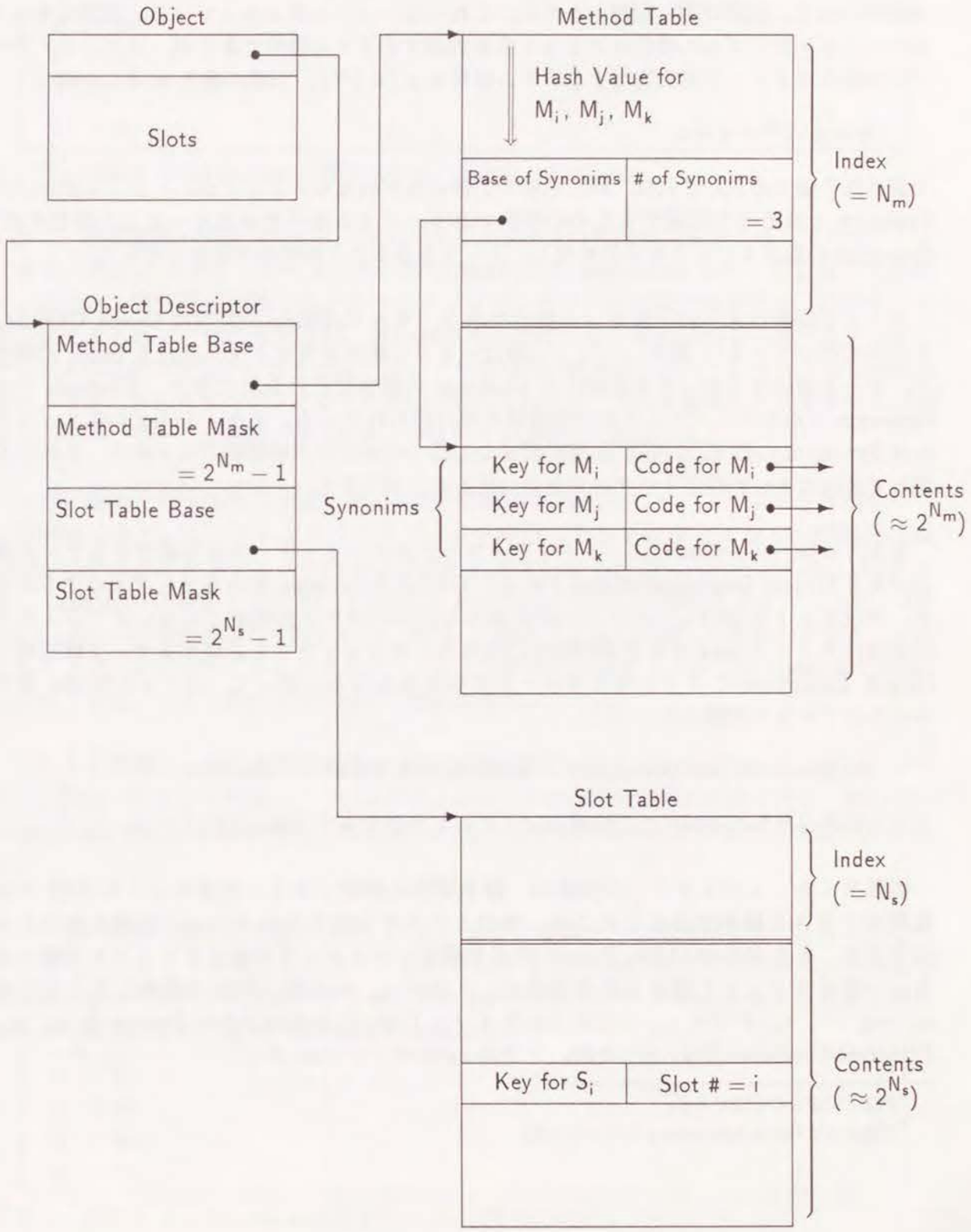


図 A-1: オブジェクト / クラスの表現

アドレス及びスロットの相対位置の対応表である，メソッド・テーブルとスロット・テーブルが作られる。探索時間の短縮のために，これらはハッシュ表となっている。探索のキー k はスロット・テーブルの場合はスロット名を表現するアトム番号であるが，メソッド・テーブルの場合はメソッド名を表現するアトム番号を a $(< 2^{24})$，引数の数を n とした時；

$$k = n \cdot 2^{24} + a + n$$

で表される値である。これは，同じ名前で引数の数が異なるメソッドが，ハッシュ表の上でSynonym となるのを回避するための手法であり，アトム番号のみをキーとした場合には* Synonym の数が 3 ～ 4 となるのに対し，1 ～ 2 となることが明らかになっている。

ハッシュ関数はキーの下位ビット抽出であり，また Collision の対策は Direct Chaining を用いて行っている†。即ち，ハッシュ表はハッシュ値によりアクセスされる Index の部分と，キーと対応するデータを格納した Contents の部分に分けられており，Contents では Synonym となるキー／データが連続領域に割付けられている。また，Index の各エントリには Synonym の数と，対応する連続領域の先頭へのポインタが格納されており，これらを用いて高速にキーに対応するデータを見つけることができる。

また，コンパイル時にはメソッド・テーブルとスロット・テーブルを結合するデータ構造である Object Descriptor が生成される。Object Descriptor には各テーブルのアドレスと，下位ビットを抽出してハッシュ値を求めるためのマスクが格納される。オブジェクトの生成はメソッド:new により実行時に行われる。オブジェクトを表現するデータ構造は，Object Descriptor のアドレスとスロットの集合からなる。従って，メソッド呼出しやスロットのアクセスの際には；

Object → Object Descriptor → Method/Slot Table → Code/Slot

という探索がなされるが，この処理はマイクロプログラムで高速に実行される。

処理系にとってのもう一つの問題は，継承関係の解決である。前述のように ESP では複数のクラスを継承することにより，他のクラスで定義されたメソッドを取り込むことができる。また Before/After Demon により継承したメソッドを修正することも可能である。一般にメソッドに関する継承関係は，クラス c_0 が直接／間接に継承したクラスを c_1, c_2, ... c_n, クラス c_i で定義されたメソッド m の Before/After Demon を b_i, a_i, Principal Predicate を p_i とした時，クラス c_0 のメソッド m が；

*PSI-I ではこの方法である。

†PSI-I では Open Addressing を用いている。


```
m :- b0, b1, ..., bn, p, a0, a1, ..., an.
p :- p0.
p :- p1.
    ⋮
p :- pn.
```

という *Method Combination* に等価である，と定められている。

さて，継承関係の解決方法には，コンパイル／リンク時に継承した全てのクラスのメソッドをまとめてメソッド・テーブルや上記の Method Combination を作ってしまう「静的な」方法と，実行時に継承したクラスのメソッド・テーブルを順次探索する「動的な」方法とが考えられる。静的な方法は，実行時間の面ではかなり有利である一方，メソッド・テーブルが他のクラスで定義されたメソッドを含むために大きくなってしまうという欠点がある。また，あるクラスでメソッドの追加／削除を行うと，それを継承しているクラスの再コンパイル／リンクが必要となるのも，プログラム開発上の問題である。

一方，動的な方法の欠点である実行時間の問題の解決法として，メソッドを呼出すコードの直後（など）に，どのクラスのメソッドを呼出したかを記憶しておく，所謂 In-line Cache の手法が，オブジェクト指向の言語の処理にしばしば用いられる。しかし ESP では，Demon が存在することと，Principal Predicate が OR の関係にあることから，In-line Cache の有効性が極めて疑わしい。即ち，一回のメソッド呼出しで複数のクラスで定義されたメソッドを実行することがあり，これを効率的に処理するためには Method Combination の存在，つまり静的な継承関係の解決が不可欠となる。

これらを勘案し，PSI 系列の処理系では静的な方法を選択することとした。但し，プログラム開発の容易さを考慮して，プログラムのロード時に継承関係の解決を行う一種の Dynamic Link 方式を採用し，部分的な修正が他のモジュールに波及することを防止している。

この他に継承に関する問題として，メソッド中でのカットの処理がある。メソッド中のカットは，メソッドが属しているクラスに存在する Alternative だけではなく，Method Combination で結合された他のクラスの Pricipal Predicate も Alternative とみなして除去しなければならない。即ち，Method Combination が；

```
m :- p.
p :- p0.
p :- p1.
p :- p2.
```

であり，クラス c_0, c_1, c_2 における p_0, p_1, p_2 の定義が；

```
p0 :- ..., !, ...
p0 :- ...          ×
p0 :- ...          ×

p1 :- ...          △
p1 :- ...          △

p2 :- ...          △
p2 :- ...          △
```

である時，p_0 の第一クローズのカットは，× を付したクローズだけではなく，△ を付したクローズも除去しなければならない。

これを実現するために，PSI-I では後述する遠隔カットを用いて，メソッド中のカットは 1 レベル上の Alternative をカットするようにしている。しかし遠隔カットはカット後に最新となる Choice Point を順次探索する「重たい」処理を含むため，PSI-II, PSI-III では 2.3 で述べた [Debray 86] に似た手法を用いて高速化を図っている。即ち，Method Combination を；

```
m :- load_choice_point(C), p(C).
p(C) :- p0(C).
p(C) :- p1(C).
p(C) :- p2(C).
```

として，カット後に最新となるべき Choice Point のアドレスを引数として p_i に渡す。これを用いて，メソッド中のカットには；

```
p0(C) :- ..., method_cut(C), ....
```

のように，最新となるべきの Choice Point を陽に与えて，Choice Point の順次探索を省いている。

これらの手法を用いて，PSI-II では append を，メソッドにより実現したものと通常の述語を用いたもの，即ち；

- Method Version

```
:append(Obj,[],L,L):- !;
:append(Obj,[X|L1],L2,[X|L3]):- :append(Obj,L1,L2,L3);
```

- Local Predicate Version

```
append([],L,L):- !;
append([X|L1],L2,[X|L3]):- append(L1,L2,L3);
```

の性能比を，1:2 以下にすることができた [Ikeda 87]。

A.1.4 ヒープ

ESP のスロットは名前による「大域的な」アクセスが可能である他，状態変数としての機能，即ち「更新」することができるという Prolog の変数にはない機能を持っている。また，この更新操作は「副作用」として行われるため，バックトラック時にも更新後の値を保持することができるという特徴を持っている。

このような「非論理的な」変数を，通常の「論理変数」と同様に取扱うことは，言語のわかりやすさという点で問題であるだけではなく，処理の効率の面からも好ましい方法ではない。即ち，LISP の rplaca, rplacd, setf などの操作を Prolog のリストや構造体に適用すると，バックトラックによってグローバル・スタックを縮めるという効率的な記憶管理を行うことができなくなる。

そこで，スロットを含んでいるデータ構造であるオブジェクトは，グローバル・スタックとは別の領域である「ヒープ」*に割付けられる。また，ヒープに割付けられるデータ構造として，命令コード，更新可能な構造体であるヒープ・ベクタ†，文字列なども用意されている。

これらのデータは全て動的な生成と更新が可能である一方，バックトラックによる消去や復元は行われない。従って，ヒープはガベージ・コレクション以外では，その領域が縮小することはない。

また，ヒープ上のデータ構造の要素に，未定義変数や通常の構造体を代入すると，バックトラックの後に「存在しない」データがヒープ上に残ることになる。従ってこのような代入は禁止されており，ヒープ・データの要素を更新する組込述語によってチェックされる。なお，未定義変数や通常の構造体をヒープ上に凍結する組込述語 freeze や，それを元に戻す melt が用意されており，ヒープとスタック間でのデータの変換を行うことができる。

*[Warren 83] の Heap はグローバル・スタックに相当し，ここでいうヒープとは異なる。
†KL0 では通常の構造体をスタック・ベクタと呼ぶ。

A.2 実行順序制御

前述の，非数値に対する加算が行われた際のエラー処理，即ち；

(1) エラーの検出
(2) ユーザへの通知
(3) open されたファイルの close などの「後始末」
(4) トップ・レベルへの大域脱出

は，以下に示す KL0 の実行順序制御機能を用いて実現することができる。

(1,2) 組込述語によるエラー検出と例外 (Exception) の発生。

(3,4) 遠隔カットとバックトラックによる大域脱出，及び On-Backtrack による「後始末」ルーチンの起動。

以下，各々の具体的な機能や処理方式について述べる [Yokota 84, Nakashima 87d]。

A.2.1 例外 (Exception)

前述の例では，エラーの検出は加算を行う組込述語 `add(X,Y,Z)` により行われる。即ち，`add` の第一，第二引数は整数または浮動小数点数でなければならないという「制限条件」があり，これに違反した場合には例外 (Exception) が発生する。KL0 の組込述語の多くは様々な制限条件を持っており，例えば構造体 V の N 番目の要素 E を取り出す組込述語 `vector_element(V,N,E)` は；

- V がスタック・ベクタまたはヒープ・ベクタである。
- N が整数である。
- N が 0 以上，かつ V の要素数未満である。

という制限条件を持っている。

さて例外を検知すると，組込述語は例外の種類に応じてあらかじめ定義された「例外ハンドラ述語」の呼出しに置換される。このハンドラは引数として，組込述語の種類，例外の種類と付随する情報，組込述語に渡された引数を要素とするスタック・ベクタ，及び組込述語のアドレスが渡される。例えば；

```
p(X,Y,Z):- add(X,Y,Z), ...
```

で X が非数値である場合には；

```
p(X,Y,Z):- exception_handler(add,illegal_input,1,{X,Y,Z},Addr),
           ...
```


に置き変わる*。このハンドラを；

```
exception_handler(Pred, Type, ErrArg, Args, Addr):-
        error_message(Pred, Type, ErrArg, Args, Reply),
        error_action(Reply).
```

のように定義しておけば，エラーの通知と，ユーザからの返答に基く処置を行うことができる。

このように，例外発生時にハンドラを呼出す機構により，例外処理の統一性と柔軟性ともに満足させることができる。例えば，DEC-10 Prolog [Bowen 77] では，非整数の加算は；

- コンパイル・コードの実行時はエラー。
- インタプリタでは引数が数式であれば実行，そうでなければエラー。

であり，また構造体の要素を取り出す `arg(N,V,E)` は，前述と同様の条件を満足していないと失敗する。このような不統一さはプログラマを混乱させ，デバッグの効率を低下させる要因となるため，KL0 ではデフォルトのハンドラとしてエラー処理を行うものが定義されており，全ての例外を統一的に取扱っている。

一方，プログラマがハンドラを定義することを可能にすることによって，例外処理の柔軟性を実現している。ハンドラの再定義は組込述語 `exception_hook(Type,Handler)` によって行われ，例えば；

```
..., exception_hook(illegal_input, my_handler/5),...,
        ⋮
my_handler(vector_element, illegal_input, _, _, _):- !, fail.
my_handler(P,T,AN,AV,AD):- default_handler(P,T,AN,AV,AD).
```

とすることにより，`vector_element` についてのみ，DEC-10 Prolog の `arg` と同じ処理とすることができる。また；

```
my_handler(add, illegal_input, _, {X,Y,Z}, _):-
        evaluate(X, X1), evaluate(Y, Y1), add(X1, Y1, Z).
my_handler(subtract, illegal_input, _, {X,Y,Z}, _):-
        evaluate(X, X1), evaluate(Y, Y1), subtract(X1, Y1, Z).
        ⋮
evaluate(X, X):- integer(X),!.
evaluate(X+Y, Z):-!,
        evaluate(X, X1), evaluate(Y, Y1), add(X, Y, Z).
evaluate(X-Y, Z):-!,
        evaluate(X, X1), evaluate(Y, Y1), subtract(X, Y, Z).
        ⋮
```

*実際は `add` や `illegal_input` はアトムではなく整数の識別子である。

によって，DEC-10 Prolog のインタプリタと同様の数式処理も可能となる。

なお，例外には組込述語が発生させるものの他，ユーザ定義例外，トレース例外，及び割込例外がある。ユーザ定義例外は，`exception_hook(Type,Handler)` でシステムが用意したもの以外の `Type` を指定することにより，そのハンドラとともに定義することができる。この例外を発生させるためには，組込述語 `raise(Type,Args)` を実行すれば良い。また，システムが用意した例外も `raise` によって陽に発生させることができ，プログラムの中で検出した異常を組込述語が検出したものと統一的に扱うことができる。

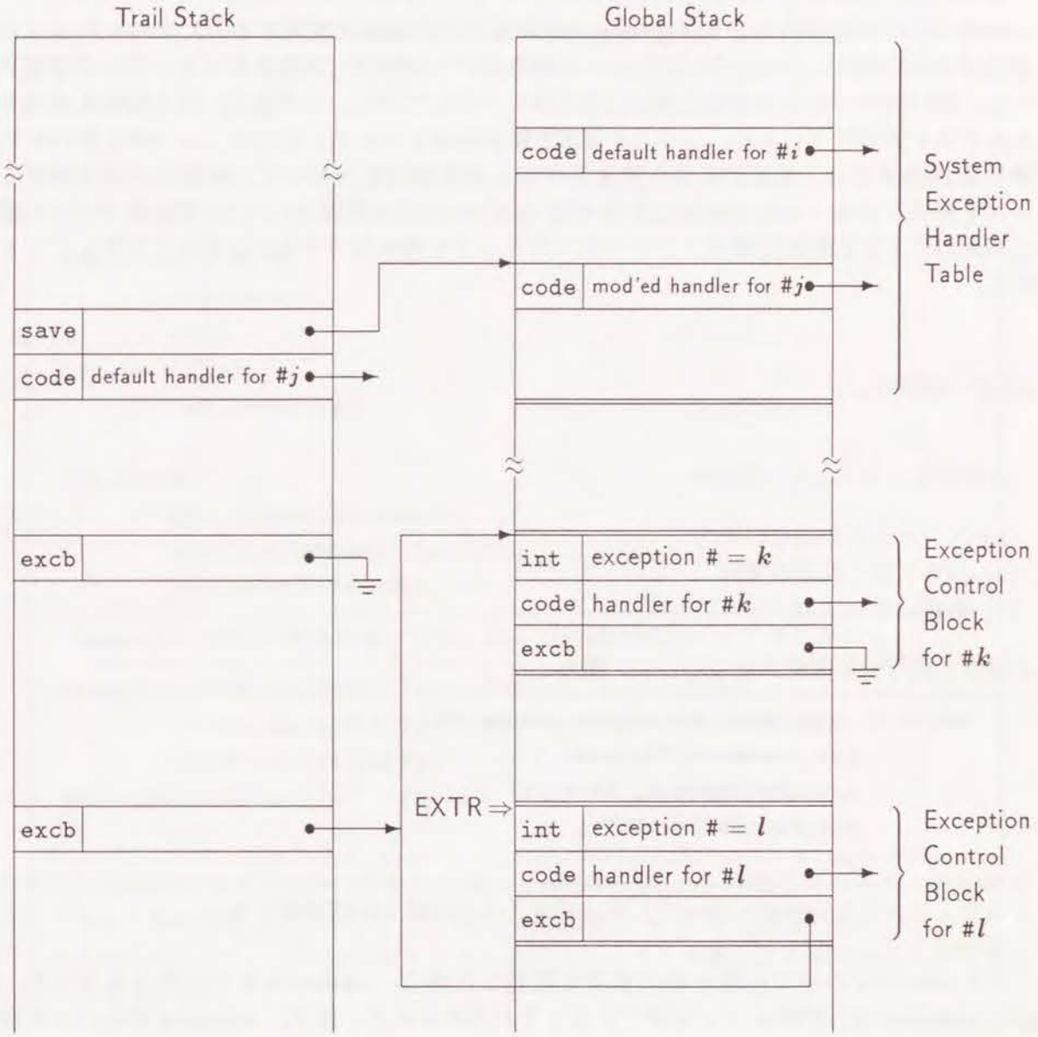
トレース例外はプログラムのデバッグのために使用され，トレース・モードでの実行が指示されると，全てのゴール（組込述語を含む）がトレース例外を発生する。トレース例外のハンドラにはゴールの種類，呼出そうとしている述語のアドレス，引数などが渡され，これに基き DEC-10 Prolog などのデバッグ機能である creap, skip, leap や，述語名／引数などの表示を行うことができる。この結果，通常の処理系ではインタプリティブ・コードに対してのみ可能なデバッグ・サポートを，コンパイル・コードに対して適用し，デバッグ対象のプログラムの高速実行を実現することができた。

また割込例外は，例えば Control-C の入力によるプログラムの中断などのために用いられる。この場合，実行中のプロセスの外部（例えばキーボード入力割込ハンドラ）からの要求により，プログラム中に例外ハンドラが（置換ではなく）挿入される。例外ハンドラは普通の例外処理と同様に，中止／続行の問い合せ，その結果に基く大域脱出などを行う。従って外部要因による例外処理を，内部要因によるものと統一して取扱うことができる。

さて，例外の処理は図 A-2に示すように行われる。まず例外ハンドラの登録／変更は，システム定義のものとユーザ定義のものとが，別々の方法で処理される。32 種類のシステム定義例外のハンドラを呼出すための情報，即ちハンドラのコード・アドレスは，グローバル・スタックのボトムに置かれたテーブルに登録される。また，このテーブルはシステムで用意した，デフォルトのハンドラのアドレスに初期化される。

`exception_hook` によるハンドラの再定義がなされると，ハンドラ・アドレスの旧値がテーブルのエントリ・アドレスと共に，トレイル・スタックにプッシュされる。バックトラック時には，この情報に基きハンドラの復元が行われる。従って，後述のバックトラックを用いた大域脱出などの際に，ハンドラの変更を心配する必要がない。なお，トレイル・スタックにプッシュされた情報が，通常の変数アドレスとは違ってアドレス／データのペアであることを知るために，アドレスに特殊なタグである `save` が付される。また，カット操作における Tidy Trail の際も `save` を認識して，除去の対象外とする。

組込述語が例外を検知すると，例外の種類に基き上記のテーブルをアクセスしてハンドラ・アドレスを求め，引数を適宜生成して分岐する。なお，2.4 で述べた First Goal Opti-



- System exception $\#j$ is modified.
- Two user exeptions, $\#k$ and $\#\ell$, are defined.

図 A-2: 例外の処理

mization を組込述語に拡張した適用を行う場合，例外発生による組込述語の「通常述語」（即ち例外ハンドラ）への置換の可能性に注意しなければならない。この問題については，トレース例外や割込例外の効率的な処理手法とともに，4.3 で論じている。

一方，ユーザ定義例外を exception_hook によって登録／変更すると，グローバル・スタック上に，例外とハンドラ・アドレスからなるデータ構造（例外制御ブロック）が生成される。例外制御ブロックは生成された順にリンクされており，その先頭，即ち最後に生成されたブロックのアドレスは，レジスタ **EXTR** が保持している。従って，ユーザ定義例外に対する raise では，**EXTR** から始まるブロックの連鎖をたどって，例外ハンドラのアドレスを得る。なお，exception_hook によるブロック生成時には，バックトラックでの復元のために **EXTR** の旧値が，トレイル・スタックに特殊なタグ excb を付してプッシュされる。

A.2.2 遠隔カット

対話的なシステムは一般的に；

(a) ユーザからの指令の入力
(b) 指令に基く処理の実行
(c) 処理結果の出力

を繰返し実行する構成となっている。即ち；

```
while (! top_level_terminate_condition) {
        get_commmand(Command) ;
        execute(Command, &Result) ;
        put_result(Result) ;
        }
```

という「トップ・レベル・ループ」が，プログラムの最上位に存在する。

さて execute の中で，何らかの異常を発見した場合，execute を中止するとともに，get_command から次のループを続行することが求められる。即ち，execute からの「大域的」な脱出と，それをトップ・レベル・ループという，ある一定の処理レベルに留めることが必要である。

この「大域脱出」はもちろん，通常の脱出方法であるサブルーチンからの復帰と，呼出し元での終了状態のチェックにより，実現することも可能である。しかし，この方法はプログラミングが繁雑であるばかりではなく，特に呼出／復帰が頻繁に行われる関数型や論理型の言語では，致命的な性能低下を招く。また，所謂 goto による実現も考えられるが，プログ

ラムの「美しさ」が損なわれることや，呼出／復帰を制御するスタックを縮めることができないなど，好ましい方法ではない。

そこで，プログラム言語の中に複数レベルの復帰を行う機能を組込むことが考えられる。例えば Common LISP などでは，脱出点を定義する catch と，そこへ直接復帰する throw が用意されている。また C においても，setjmp と longjmp を用いて，同様の機能を実現することができる。

一方論理型言語では，呼出／復帰の機構とは別に，バックトラックという制御機構があり，これを大域脱出に用いることが考えられる。例えばトップ・レベル・ループを；

```
top_level_loop:-
        top_level,
        fail.
top_level_loop:-
        top_level_loop.

top_level:-
        get_command(Command),
        execute(Command, Result),
        put_result(Result), !.
```

とし，execute の中の異常処理ルーチン，例えば前述の例外ハンドラにおいて；

```
exception_handler(Pred, Type, ErrArg, Args, Addr):-
        error_message(Pred, Type, ErrArg, Args, Reply),
        error_action(Reply).
error_action(abort):-!, fail.
    ⋮
```

のように強制的にバックトラックを行えば，top_level_loop の第二クローズへの分岐が実現できるように見える。なお，この方法での大域脱出は，呼出／復帰のためのローカル・スタックだけではなく，グローバル・スタックやトレイル・スタックも縮めることができるという利点がある。

しかしこの方法は，大域脱出を行う時点で execute の中に未実行の OR 分枝が存在すると，それを実行してしまうという重大な欠点がある。これは，前述の ESP-Listener のように execute の中味を特定できないようなシステムでは致命的であるし，一般のシステムにおいても大域脱出の際に OR 分枝が存在しないことを保証するのは困難であることが多い。

さて，execute（及び put_result）が完了した場合には，top_level の末尾におけるカットによって全ての OR 分枝が除去されるため，top_level_loop の第一クローズにお

ける fail は不要な OR 分枝へのバックトラックを行わない。従って，error_action が fail の前に同じようなカットをできれば，バックトラックによる大域脱出を実現することができる。このために用意されたのが遠隔カットである。

KL0 ではどのような OR 分枝をカットするかの指定を，「呼出レベル」という概念を用いて行う。呼出レベルは，述語を呼出すと 1 だけ増え，また復帰すると 1 だけ減るカウンタであり，組込述語 level(Level) によって，それが含まれるクローズの呼出レベルを知ることができる。遠隔カットを行う組込述語 absolute_cut(Level) は，それが含まれるクローズの「直系の祖先」であり，かつその呼出レベルが Level よりも大きいような OR ノードの分枝をすべて除去する。従って，前述の top_level_loop などの例は；

```
top_level:-
        level(Level),
        store_abort_level(Level),
        get_command(Command),
        execute(Command, Result),
        put_result(Result), !.
    ⋮
error_action(abort):-
        get_abort_level(Level),
        absolute_cut(Level),
        fail.
    ⋮
```

のように記述することができる*。

なおこの他に，相対的な呼出レベル（例えば 3 レベル上）によるカットを行う relative_cut(Level) や，複数レベルの復帰を行う succeed(Level) も用意されている。

さて，これらの機能の実現のために，処理系ではレベル・カウンタ (**LVLC**) なるレジスタを用意している。**LVLC** のインクリメントは呼出しの際に行えば良いが，デクリメントに関しては TRO によって複数レベルの復帰が行われるため単純ではない。そこで Environment に **LVLC** を退避し，復帰の際に（即ちユニット・クローズの実行完了時）復元するようにしている。また，バックトラック時にも復元が必要であるため，**LVLC** は Choice Point にも退避される。

Environment 及び Choice Point に退避した **LVLC** の値は，遠隔カットの処理のためにも用いられる。即ち，遠隔カットにより最新となるべき Choice Point は以下の方法により

*store_abort_level や get_abort_level はオペレーティング・システム SIMPOS の機能を用いて実現することができる。

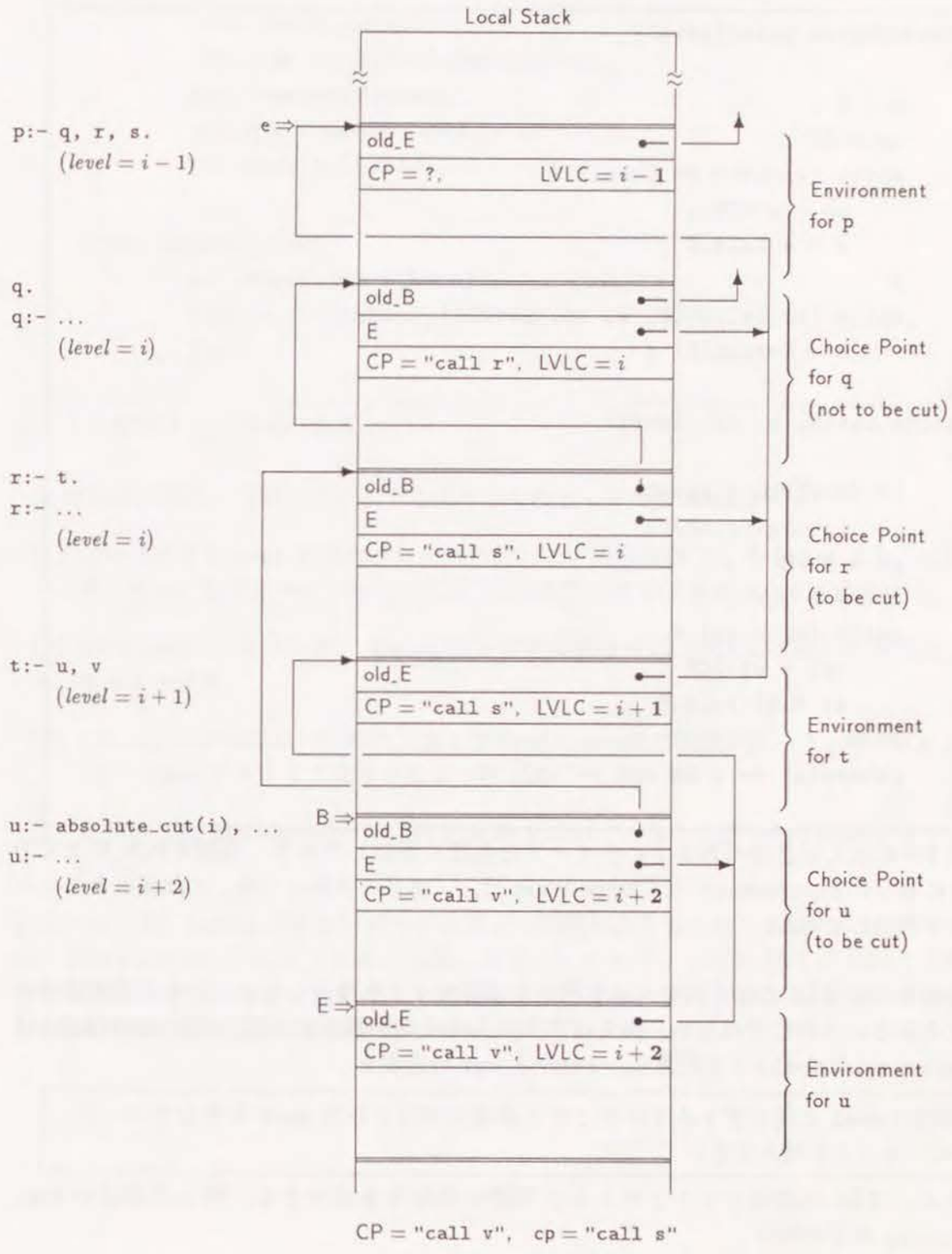


図 A-3: 遠隔カットの処理

見つけることができる（図 A-3）。

```
find_choice_point(level)
{
    e = E ;
    cp = CP ;
    while (e->LVLC >= level) {
        cp = e->CP ;
        e = e->old_E ;
    }
    while (to_be_cut(B, e, cp, level))
        B = B->old(B) ;
}
to_be_cut(B, e, cp, level)
{
    if (B->LVLC < level)
        return(FALSE) ;
    e1 = B->E ;
    cp1 = B->CP ;
    while (e1 > ee) {
        cp1 = e1->CP ;
        e1 = e1->old_E ;
    }
    return(e1 == e && cp1 == cp) ;
}
```

これは 2.3 に示した普通のカットのナイーブな処理に類似しており，退避された **E** と **CP** がともに等しい Environment と Choice Point は，「直系の先祖と子孫」の関係にあるということを利用している。

以上のように KL0 では呼出レベルを用いた遠隔カットを導入したが，この方式が最善のものであるというわけではない。例えばアトム Label を引数とする組込述語 `mark(Label)` と `remote_cut(Label)` とを用意し，`remote_cut` の定義を；

> 引数 `Label` と同じアトムを引数とする最後に実行された `mark` を含むクローズに，カットを挿入する。

とすれば，KL0 の遠隔カットとほとんど同様の機能を実現できる。例えば前述の `top_level_loop` などの例は；


```
top_level:-
        mark(abort_label),
        store_abort_label(abort_label),
        get_command(Command),
        execute(Command, Result),
        put_result(Result), !.
    ⋮
error_action(abort):-
        get_abort_label(Label),
        remote_cut(Label),
        fail.
    ⋮
```

のように記述することができる。

この方式の場合，実現手法は以下のようなものとなる（図 A-4）。

(1) コンパイラは `mark` を含むクローズに対して，局所変数 $\mathbf{Y_1}$ を `Label` のために確保する。また，このクローズについては TRO を行わないようにコード生成をする。

(2) `mark(Label)` は $\mathbf{Y_1}$ に，`Label` のタグを特殊なもの（例えば `lab`）に置き換えた値をセットする。

(3) `remote_cut(Label)` は **E** から始まる Environment の連鎖から，$\mathbf{Y_1}$ のタグが `lab` で値が `Label` であるようなものを見つける。

(4) 見つかった Environment が保持する **B** を用いて，2.3 で述べた手法によりカットを行う。

この方法では，`Label` の重複回避などに若干の問題があるものの，KL0 での述語呼出／復帰，Environment/Choice Point の生成，及びバックトラック時の **LVLC** に関する操作を完全に排除でき，かなりの性能向上を望むことができる。現時点での KL0 仕様の改変はかなり困難を伴うが，重要な検討課題の一つであろう。

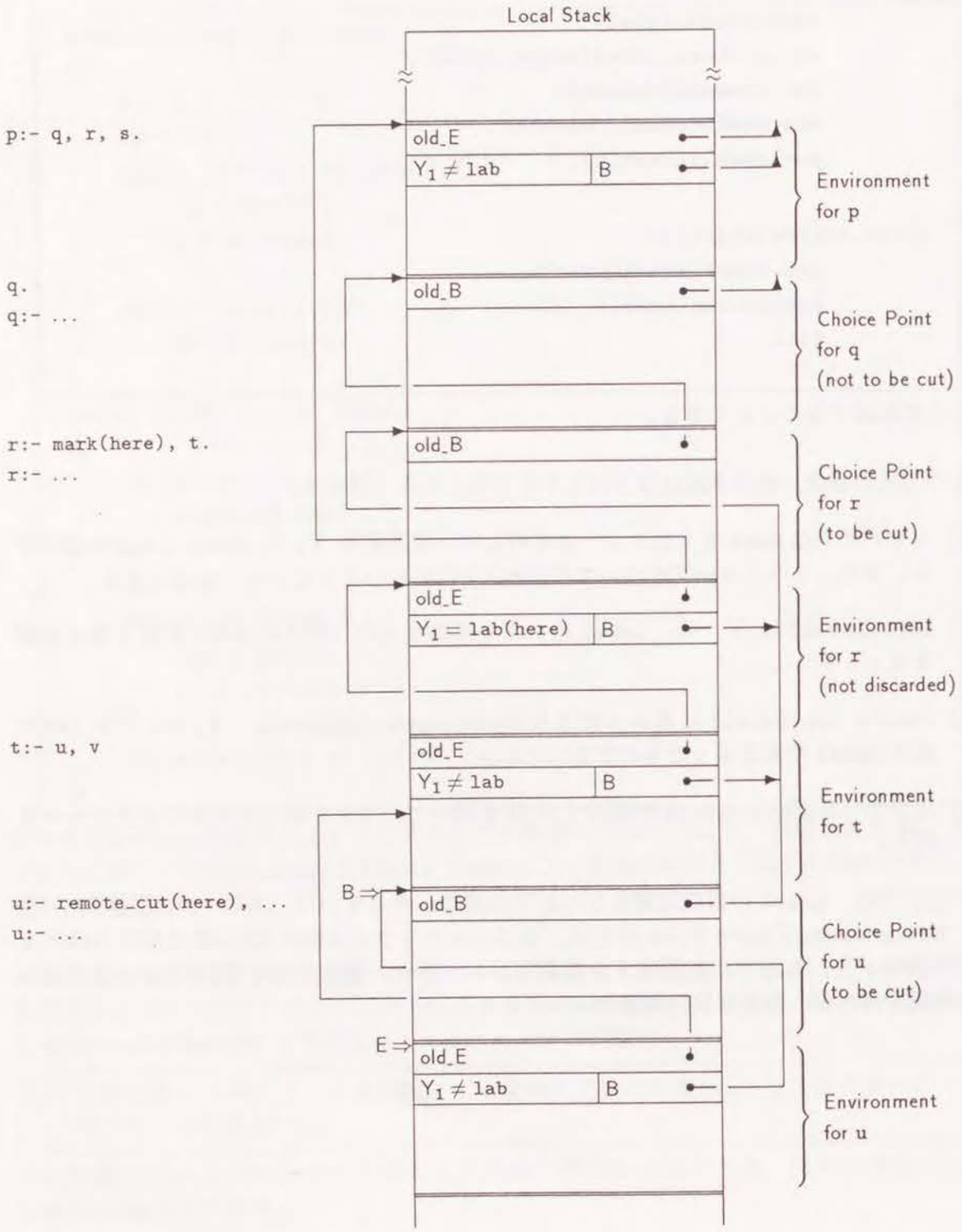


図 A-4: mark による遠隔カットの処理

A.2.3 On-Backtrack

エラー検出時のアボート処理などの際に，それまでに行った処理の「後始末」を求められることがしばしばある。例えば前述の ESP-Listner では，トップ・レベル・ループへの大域脱出時に，open されているファイルを全て close してしまうことが望まれる。

例えば Common LISP などでは，`unwind-protect` という機構があり；

```
(unwind-protect expression clean-up)
```

によって，*expression* が終了または大域脱出した際に，後始末処理 *clean-up* の実行を指示することができる。また，この機構を用いた；

```
(with-open-file file-spec expressions)
```

は，*file-spec* で定まるファイルを open した後 *expressions* を `unwind-protect` を用いて実行し，その *clean-up* でファイルの close を行う。

一方 KL0 では，On-Backtrack という機構により，後始末処理を実現することができる。組込述語 `on_backtrack(Goals)` は；

```
on_backtrack(Goals):- true.
on_backtrack(Goals):- call(Goals), fail.
```

にほぼ等しく，バックトラックの際に実行すべきゴール列を指定する機能を持つ。但し，このゴール列を呼出すための OR 分枝は，カットによっても除去されないという特徴があり，遠隔カットとバックトラックによる大域脱出の際にも必ず実行される。従って，Common LISP の `with-open-file` に相当する機能を；

```
with_open_file(File, Procedure):-
        open(File), on_backtrack(close(File)), call(Procedure),
        close(File).
```

により実現することができる。またさらに一般的に；

```
open_file(File):- open(File), on_backtrack(close_if_open(File)).
close_if_open(File):- opened(File), close(File).
```

として，大域脱出時の close のみを指定することもできる。

この他，ESP のスロット代入をバックトラック時に Undo する操作も，`on_backtrack` を用いて以下のように実現できる。

```
set_slot_with_undo(Object, Slot, Value):-
        Old_Value = Object!Slot, Object!Slot := Value,
        on_backtrack(undo_slot(Object, Slot, Old_Value)).
undo_slot(Object, Slot, Old_Value):- Object!Slot := Old_Value.
```

また二引数の on_backtrack(Goals, ID) もあり，On-Backtrack 指定された Goals にユニークな ID が与えられる。この ID を用いて reset_on_backtrack(ID) により，On-Backtrack 指定の解除（即ち一種のカット）を行うこともできる。なお，コンパイラは一引数の on_backtrack(Goals) 及び二引数の on_backtrack(Goals,ID) を；

```
on_backtrack(Code, Args, _)
on_backtrack(Code, Args, ID)
```

に変換する。但し，Code は Goals を呼出すためのコード・アドレスである。また，Args は Goals に出現する引数をまとめたスタック・ベクタであり，Code の引数となる。即ち；

```
Code(Args):- Goal1, Goal2, ..., Goaln, fail.
```

なるクローズが自動的に生成される。

On-Backtrack に関する処理は，図 A-5に示すような方法によって行われる。まず，組込述語 on_backtrack(Code,Args,ID) は，グローバル・スタック上に Code と Args へのポインタ，及び ID からなる制御ブロックを生成する。ID はグローバル・スタック上に存在する，制御ブロックの数に等しい整数である。また，この制御ブロックには on_backtrack が実行された時点でのグローバル・スタックのバックトラック・ポイント，即ちレジスタ **GB** の値が保存され，**GB** 及び最新の Choice Point の **G** は，制御ブロックの直下のアドレスに更新される。この操作は，Args に含まれる変数をトレイルするために行われる。

更に on_backtrack は，トレイル・スタックにも 2 ワードの制御ブロックをプッシュする。一つはグローバル・スタック上の制御ブロックへのポインタであり，バックトラックや Tidy Trail の際に特殊な取扱いが必要であることを示すタグ obcb が付される。もう一つは，トレイル・スタック上の制御ブロックを結ぶリンクであり，その先頭はレジスタ **ONTR** が保持する。このリンクは reset_on_backtrack が除去すべきものを探索するために用いられる。

さて，バックトラック時の Undo 処理の途中でタグ obcb を発見すると，Undo 処理を中止し，グローバル・スタック上の制御ブロックにしたがって Code の呼出を行う。その際，制御ブロックに保存された **GB** の復元を行う。

一方，カット処理における Tidy Trail では obcb は除去されず，単にトレイル・スタック上を移動する。但しその際，移動に伴うリンクの付け替えが行われる。また，グローバル・スタック上の制御ブロック中の **GB** の値は，カットによって最新となった Choice Point に対応して変更される。

以上のように，On-Backtrack の実現のためには，Undo 及び Tidy Trail におけるトレイル・スタック上のデータのタグ判定が必要である（例外の処理に関連する save や excb も同様）。従って，これらの機能の導入による，「通常の」バックトラックやカットの性能低

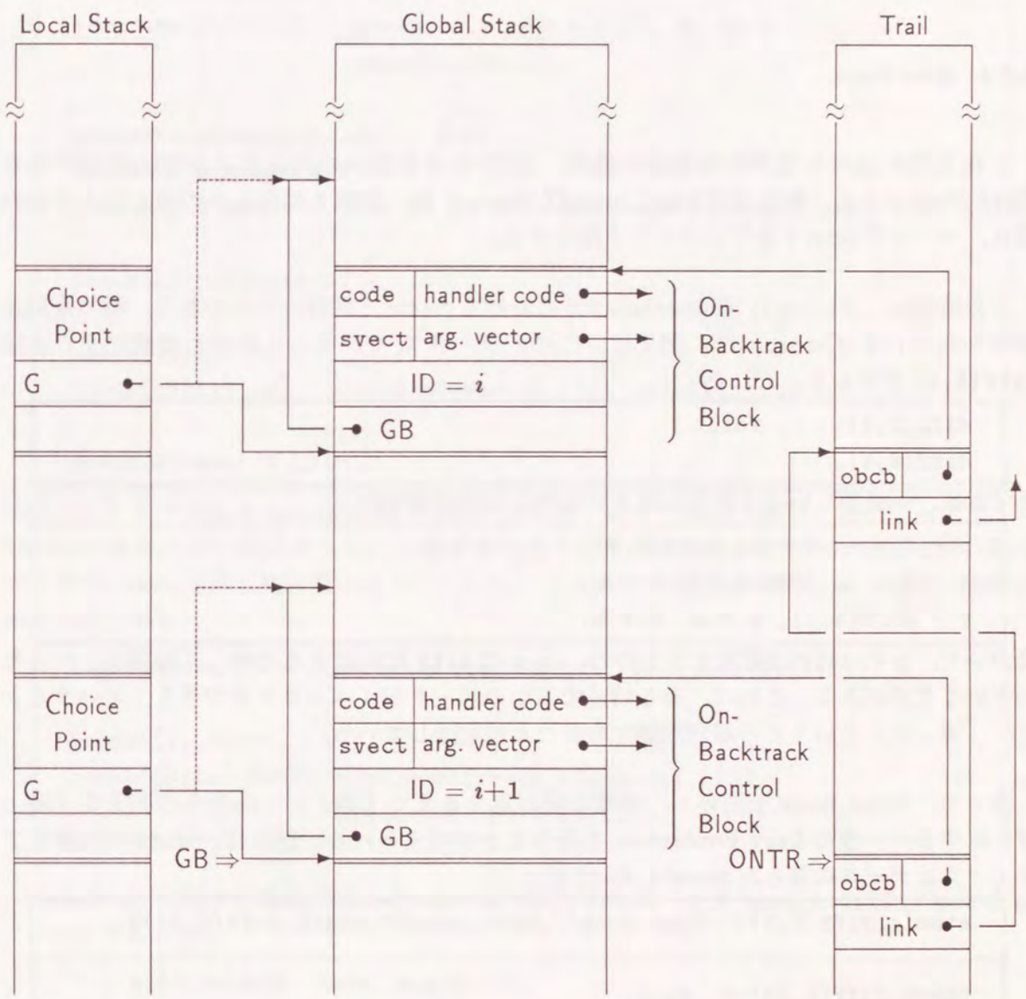


図 A-5: On Backtrack の処理

下には，充分な注意が必要である。しかしPSI系列の処理系では，タグ判定のためのハードウェア機構と，注意深くコーディングされたマイクロプログラムによって，オーバヘッドを極めて小さいものとすることができた。また4.2で示したように，Tidy Trailでは別の意味でタグ判定が有効であることや，「簡単な」バックトラック／カットでは完全にオーバヘッドを除去できることも明らかになっている。

A.2.4 Bind Hook

これまでに述べた順序制御機能の他に，KL0には変数への代入による呼出機能である*Bind Hook*がある。組込述語`bind_hook(X,Goals)`は，変数`X`に何らかの値が代入された際に，ゴール列`Goals`を呼出すことを指定する。

この機能は，Prolog II [Colmerauer 82]における*freeze*と同様のものであり，様々な用途が知られている[Carlsson 87]。例えば，二つのデータ`X`, `Y`が等しくなければ成功する述語`diff(X,Y)`を考える。これを；

```
diff(X,X):- !, fail.
diff(X,Y).
```

とすれば，一見正しいように思われる。しかしこの定義では；

```
p:- X = a, Y = b, diff(X,Y).
q:- X = a, diff(X,Y), Y = b.
r:- diff(X,Y), X = a, Y = b.
```

において，`p`の`diff`は成功するものの，`q`, `r`の`diff`は失敗するため，「論理的」ではないという欠点がある。これは，ある時点で二つのデータが「ユニファイできる」ということと，「等しい」ということが別問題であることに起因している。

そこで，`bind_hook`を用いて，変数の値が定まるまで「等しい」か否かの判定を「棚上げ」にする，一種のLazy Evaluationを行うことが考えられる。例えば，`diff`の対象をアトミックなデータに限った`atomic_diff`は；

```
atomic_diff(X,Y):- bind_hook(X,bind_hook(Y,check_diff(X,Y))).

check_diff(X,X):-!, fail.
check_diff(X,Y).
```

と記述することができる。また，これを一般のデータに拡張することも，さほど難しいことではない。

この他，Generation and Checkのスタイルで書かれたプログラムを*，データの一部分を

*プログラムはコンパクトで判りやすいものとなるが，一般に効率が悪い。

生成した時点で，その正当性を判定するCheckerを呼出すように変更することも，`bind_hook`を用いて比較的容易に行うことができる。例えば；

```
eight_queen(L):- generate(8,L), check(L).

generate(0,[]):-!.
generate(N,[X|L]):- generate_element(8,X), N1 is N - 1,
                    generate(N1,L).

generate_element(0,_):-!, fail.
generate_element(N,N).
generate_element(N,X):- N1 is N-1, generate_element(N1,X).

check(L):-check(L,L).

check([],_):-!.
check([X|L1],L2):- check_element(X,L2), check(L1,L2).

check_element(X,L):- ...
```

は極めてナイーブな8-queensの解法である。このプログラムの一部を，以下に示すように`bind_hook`を用いて記述すると，一要素を生成するたびに判定がなされ，無駄な解（例えば全てのqueenを同じ列に置いた`[1,1,1,1,1,1,1,1]`）の生成が抑止されるため，効率はかなり向上する。

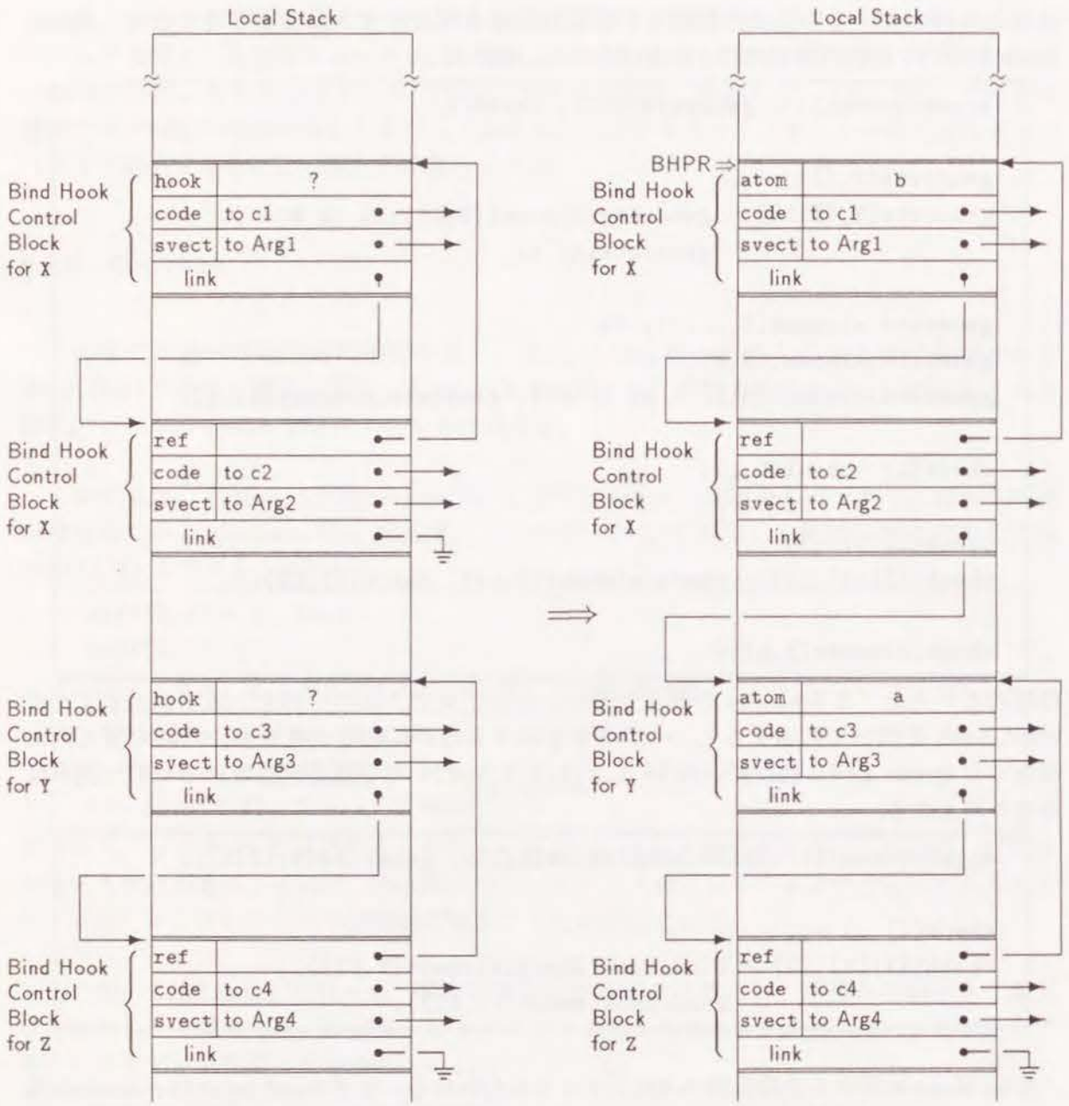
```
eight_queen(L):- bind_hook(check(L,L)), generate(8,[X|L]).

check([],_):-!.
check([X|L1],L2):- bind_hook(check_element(X,L2)),
                   bind_hook(check(L1,L2)).
```

Bind Hookに関する処理は図A-6に示すように行われる。まず`bind_hook(X, Goals)`はコンパイラにより；

```
..., bind_hook(X, Code, Args), ...
Code(Args):- Goal1, Goal2, ..., Goaln.
```

に変換される。組込述語`bind_hook`は，グローバル・スタック上に`hook`というタグを付した変数セル，`Code`, `Args`からなる制御ブロックを生成し，変数`X`には`hook`のセルへのReference Pointerをセットする。また，制御ブロックにはブロックを結合するためのリンクも含まれる。このリンクは；

- 既にHookされた変数に対する再度のHook




```
p:- bind_hook(X,c1,Arg1), bind_hook(X,c2,Arg2),
    bind_hook(Y,c3,Arg3), bind_hook(Z,c4,Arg4), q(X,Y,Z).
q(X,Y,Y):- r(Y,X).
r(a,b):- ...
```

図 A-6: Bind Hook の処理

- Hook された変数どうしのユニフィケーション
- ヘッド・ユニフィケーションでの複数の Hook された変数への代入

に用いられ，いずれの場合も複数の呼出が順次行われることになる。

ヘッド・ユニフィケーションで，Hook された変数に対して代入を行うと，対応する制御ブロックに格納された Code の呼出が行われるが，これは全てのヘッド・ユニフィケーションが完了した時点まで遅延される。従って，処理系は起動すべき制御ブロックをリンクした上でその先頭をレジスタ **BHPR** に記憶する。その後，ヘッド・ユニフィケーションが完了した時点で Hook 変数への代入がなされていれば，順次呼出を行う。

ここで問題となるのは，ヘッド・ユニフィケーションの完了と Hook 変数への代入の有無をいかに知るかである。例えば end_head なる命令を全てのクローズの ':-' に対応する部分に挿入し，そこで Hook 変数の代入を調べるのは容易である。しかしこの方法は，低頻度の処理である Hook 変数への代入のために，「通常の」処理の性能が低下するという欠点がある。また，ヘッド・ユニフィケーション完了後に動的に呼出が挿入されることは，2.4 で述べた First Goal Optimization の適用にも問題が生じる。これらの効率的な解決法については，4.3 において詳しく論じている。

付録 B　KL1

B.1　言語仕様

B.1.1　GHC の言語仕様

一般に Committed Choice Language の述語は，以下の形式で記述される。

```
H1 :- G11, G12, ..., G1n1 | B11, B12, ..., B1m1.
H2 :- G21, G22, ..., G2n2 | B21, B22, ..., B2m2.
        ⋮
Hk :- Gk1, Gk2, ..., Gknk | Bk1, Bk2, ..., Bkmk.
```

ここで，H_i はヘッド，G_{ij}, B_{ij} はいずれも並列実行されるゴールである。また，カットに相当する「コミット」('|') の左側の部分は「ガード」，右側の部分は「ボディ」と呼ばれる。このコミットの存在により，バックトラックはガード部でのみ発生することとなる。

ゴール間の同期は，未定義変数へのユニフィケーションを用いて行われる。例えば，2つのゴール g と h が共有変数 X を持ち，g が X に代入した値を h が利用して処理を行うものとする。全てのゴールは並列実行され，その実行順序はプログラム中での出現順序などでは規程されないため，g の「後で」h を実行するには何らかの同期メカニズムが必要となる。そこで，「h は X に何らかの値が代入されるまで実行しない」というメカニズムが用いられる。即ち，h において X に関するユニフィケーションが「代入」の場合，そのユニフィケーションを中断 (Suspend) することによって h の実行は抑止される。

さて，どのユニフィケーションが代入可能で，どれが代入を待つタイプかの指定は，言語によって異なっている。例えば，PARLOG ではヘッド引数に対して入力／出力の「モード宣言」を行い，入力タイプの引数のユニフィケーションは代入を待つタイプとしている。また Concurrent Prolog では，変数に対して *Read Only Annotation* という属性を付加し，そのような変数に対するユニフィケーションは代入を待つものとしている。

一方 GHC では，「ガードでのユニフィケーションは全て代入を待つ」という規則となっている。従って，ユニフィケーションのタイプの判断は，プログラマと処理系の双方にとって容易であり，プログラムの記述性と処理効率の両面で優れた仕様となっている。また，ガードでは代入が行われないため，バックトラック（ガードでのみ発生する）時に代入の無効化が不要であることも利点の一つである。

例えばキーボードからの入力と，ディスプレイへの表示の二つのプロセスは，以下のように記述される。


```
        ..., keyboard(S), display(S), ...

keyboard(S):- true | get_char(C), S = [C|S1], keyboard(S1).
display([C|S]):- true | put_char(C), display(S).
```

即ち，keyboard は get_char によってキーボードから入力された文字 C を受け取り，それを car とするリスト・セルを引数 S に代入する。同時に cdr である S1 を引数として再帰呼出を行い，次の入力に備える。

一方，display はガードで引数とリスト・セルとのユニフィケーションを行っているため，keyboard がリスト・セルを代入するまで待つこととなる。リスト・セルが代入され表示すべき文字 C を受取ると，それを put_char によって出力するとともに，cdr である S を引数として再帰呼出を行い，次の文字が到着するのを待つ。このような「ストリーム通信」を用いた並列プロセスを容易に記述できることは，GHC の重要な特徴の一つである。

B.1.2　KL1 の言語仕様

KL1 は GHC に対して以下のような変更を加えた言語である。

(1) ガードの制限
(2) 「荘園」の導入
(3) 「プラグマ」の付加
(4) 様々な組込述語の導入

まず，ガードでは一般のゴールの呼出を禁止し，組込述語の呼出のみに限定している*。このような制限を加えた Commited Choice Language は，一般に *Flat* であると呼ばれ，処理の効率化や処理系構築を容易にするために広く採用されている。即ち，バックトラック・ポイントがネストすることがないため，バックトラック処理が極めて容易なものとなる。またこの事実と，ガードでは代入を伴うユニフィケーションがないことを利用して，Prolog よりも更に高度な Clause Indexing を行うこともできる [Kimura 89]。

次に，「荘園」の導入はオペレーティング・システム PIMOS の構築のために行われた。荘園は一群のゴールの実行を管理するための単位であり；

```
execute(Goal,Control,Report)
```

により生成される。即ち Goal 及びそれが生成するゴール群が荘園に所属し，それらの中断，再開，アボートなどを，ストリーム Control を通じて外部より指令することができる。また，実行の完了やユニフィケーションの失敗などの事象が，ストリーム Report に

*組込述語であってもガードでは呼出せないものもある。

より通知される。これらの機構により，PIMOS がユーザ・プログラムの実行を管理することが可能になると同時に，ユーザ・プログラムの中での「失敗」が PIMOS を含めたシステム全体の「失敗」とならないようにすることができた。即ち，PIMOS とユーザ・プログラムは概念的には；

```
..., pimos, user_program, ...
```

のように AND の関係にあるため，user_program が失敗すると pimos も失敗することとなる。これを；

```
..., pimos, execute(user_program,C,R), ...
```

とすることにより，user_program の失敗の外部への伝播を防止することができる。

「プラグマ」は，プログラムの性能に関する記述であり，プログラムの「意味」とは独立に与えることができる。例えば優先順位指定プラグマ；

```
g@priority(P)
```

はゴール g を優先度 P で実行することを指示する。この機能は例えば評価関数を用いたゲーム木の並列探索などに有効であり，プログラムの意味を変えることなく解の発見の高速化を行うことができる。また，負荷分散指定プラグマ；

```
g@processor(P)
```

は，g をプロセッサ P で実行することを指示するものであり，並列マシンにおける負荷の均衡のための基本機能を提供している。

最後の項目である組込述語については，数値演算，比較，構造データの操作など，約 80 種類のものが用意されている。中でもストリームのマージを行う merge と，構造体の要素を更新する set_vector_element は，特筆すべきものである。まず；

```
merge(In, Out)
```

により，入力ストリーム In を出力ストリーム Out に単にコピーするプロセス，即ち；

```
merge([X|I],XO):- ture | XO = [X|O], merge(I,O).
```

と等価なものが生成される。その後，In に対してリスト・セルではなくベクタ，例えば {I1,I2,I3} をユニファイすると，このプロセスは I1, I2, I3 をマージするプロセス，即ち；

```
merge([X|I1],I2,I3,XO):- ture | XO = [X|O], merge(I1,I2,I3,O).
merge(I1,[X|I2],I3,XO):- ture | XO = [X|O], merge(I1,I2,I3,O).
merge(I1,I2,[X|I3],XO):- ture | XO = [X|O], merge(I1,I2,I3,O).
```

と等価なものに「変化」する。更に，例えば I1 に {I11,I12,I13} をユニファイすれば，I11, I12, I13, I2, I3 の 5 つのストリームのマージャとなる。しかも，マイクロプログラム

による巧妙な実現手法により [Inamura 89]，マージの手間をストリームの数に関わらず定数時間としている*。このように高速で，かつストリームの追加／削除†が容易なマージャは，ストリーム通信を基本とするシステムにとって，極めて強力な武器となっている。

構造体の要素更新は；

```
set_vector_element(OldV,N,OldE,NewE,NewV)
```

により行われる。これは構造体（ベクタ）OldV の N 番目の要素 OldE を，NewE に置き換えたベクタ NewV を生成する。従って，A.1 で述べた KL0 のヒープ・ベクタの更新とは異なり，純粋に「論理的」な操作である。この操作をナイーブに実現すると，ベクタの要素数に比例した手間を要する。しかし，後述するデータへの参照パス数の情報を用いて，OldV への参照パスが一つしかない「普通の」場合には，OldV の要素を書換えることができ，時間及び空間計算量を定数オーダとすることができる [Chikayama 87, Inamura 89]。このように，「論理的」な枠組みの中で，従来の論理型言語では極めて困難な‡構造体要素の高速な更新操作の提供したことは，論理型プログラミングの幅の拡大という点でも意義深いものである。

*組込述語を用いなければストリーム数 n に対して $O(\log n)$ の手間となる。

†nil ([]) をユニファイすると削除される。

‡例えば [Eriksson 84] の *Mutable Array* は，時間計算量は定数オーダであるが，空間計算量が更新回数に比例する。

B.2 処理方式

B.2.1 ゴールの管理

KL1 のボディのゴールは概念的には並列に実行されるが，例えば単一プロセッサ上では何らかの順序で逐次的に実行される。また，並列プロセッサにおいても，個々のプロセッサには複数の処理すべきゴールがあって，それらはやはり逐次的に実行される。但し Prolog とは違って，あるゴールの実行が完了した後に次のゴールを実行するのではないため，呼出／復帰とは異なるメカニズムが要求される。

例えば；

```
p:- true | q, r, s.
```

において，まず q を実行するが，その実行が完了した場合だけではなく，ガードでの代入待ちのために中断した場合にも，r 更には s を実行しなければならない。また，r の実行過程において q が待っている代入が行われることもあり，その際にはいずれかの時点で q の実行を再開しなければならない。このような制御を実現するために，実行可能なゴールをプールしておくことと，代入待ちのゴールを変数に「フック」しておくことが必要である。

具体的な処理は以下の手順で行われる [Kimura 87]。まずゴールの引数は，[Warren 83] と同様に引数レジスタに用意され*，ヘッド・ユニフィケーションとガードの組込述語の実行が行われる。それらが未定義変数の代入待ちを起こすことなく成功すると，最初のゴールの引数を First Goal Optimization の手法を用いて引数レジスタにセットする。一方，二番目以降のゴールについては，その引数，実行すべきコードのアドレスなどからなる「ゴール・レコード」と呼ばれる構造データを生成し，これを実行可能なゴールのプールである「ゴール・スタック」にプッシュする。そしてこの操作が完了すると，最初のゴールのためのコードへ単に分岐し，同じ処理を繰り返す。

ゴール・スタックは図 B-1に示すように優先度に対応して複数存在し，スタック内のゴールを結合するためにゴール・レコードはリンクを持っている。ゴールの完了（ボディが空であるか組込述語のみであるようなクローズの実行完了）か中断が発生すると，最高の優先度を持つ空でないゴール・スタックの先頭のゴール・レコードがポップされ，その引数が引数レジスタにセットされた後，対応するコードが実行される。

一方，ガードの実行時に代入待ちを検知すると，その変数のアドレスを「中断スタック」にプッシュして，他のクローズのガードを実行する。その結果成功するガードを持つクローズがなければ，ゴール・レコードを作り，それを中断スタックの中の全ての変数に対して

*引数レジスタを用いない方法も提案されている [Hirano 90]。

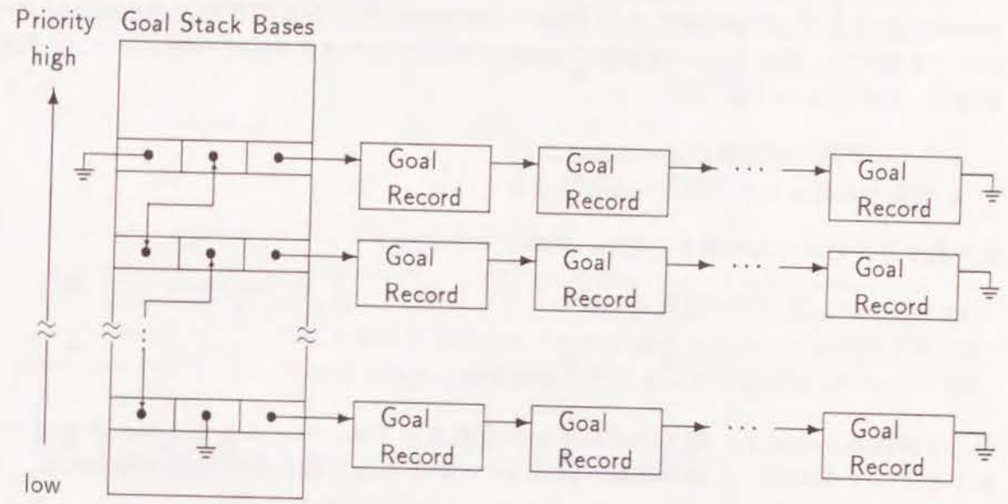


図 B-1: ゴール・スタック

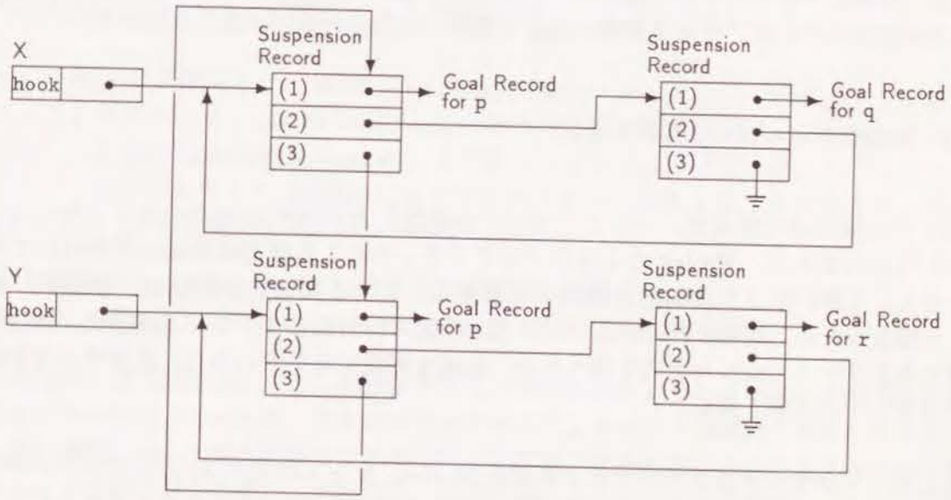


図 B-2: 中断レコード

フックする。このため，未定義変数には代入待ちのゴールの有無を示すために，タグが undef であるものと，hook であるものの二種類が必要である。また，ゴール・レコードのフック操作は，単に undef の変数を hook にするだけではなく，「中断レコード」の生成を伴う。中断レコードは；

- 1つの変数への複数のゴールのフック
- 複数の変数への1つのゴールのフック

に対処するためのものである。即ち，中断レコードは；

(1) ゴール・レコードへのポインタ
(2) 同じ変数にフックされた全てのゴールを知るためのリンク
(3) 1つのゴールが待っている全ての変数を知るためのリンク

の三つの要素から成る。例えばゴール p が変数 X と Y に，ゴール q が X に，またゴール r が Y にフックされている場合には，図 B-2に示すような二重の循環構造が作られる。そして，ボディでのユニフィケーションにより X への代入が行われると；

(a) リンク (2) をたどって p と q をゴール・スタックにプッシュし；
(b) リンク (3) をたどって Y への p のフックを解除する；

という操作が行われる。

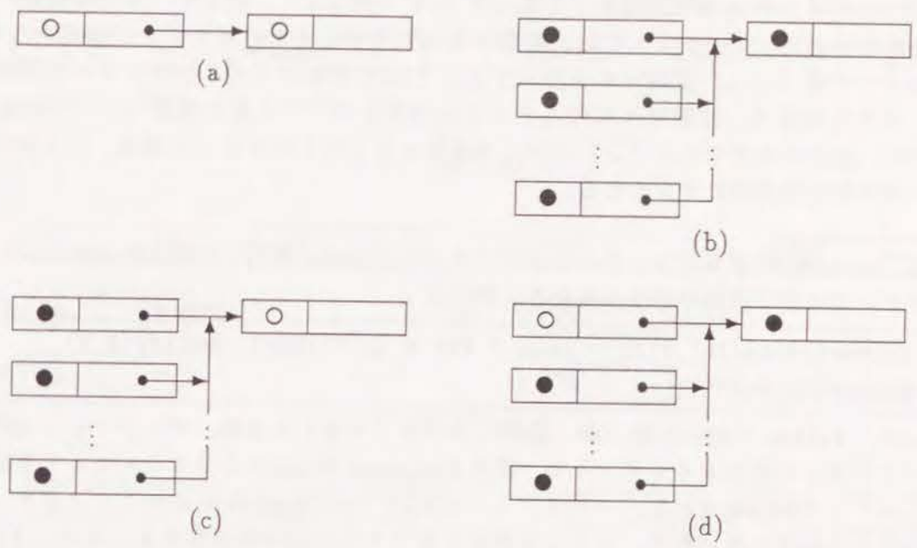
なお，ゴール・レコードや中断レコードについては，フリー・リストを用いて不要になったものの回収が行われ，メモリ消費の抑止と参照の局所性の維持が図られている。

B.2.2 実時間ガベージ・コレクション

KL1 での構造体の生成は，バックトラックが発生しないボディでのユニフィケーションでのみ行われるため，Prolog のようにバックトラックによる構造体領域の再利用はできない。また，LISP のような構造体要素の「書換え」による領域の節約法は，言語の「美しさ」を損なうため，採用されていない。一方，頻繁に行われるストリーム通信は，構造体の一種であるリスト・セルの生成を伴うため，単純な実現手法を用いると，急速にメモリが消費されることが予想される。

例えば，プロセッサ当たりのメモリ容量を 16 Mw，また1秒間に 10 万回のゴール呼出が行われ，その各々がリスト・セルを1個生成すると仮定すると*3 分以内にメモリが消費され尽くしてガベージ・コレクションが行われることとなる。このようにガベージコレクションが頻発すると，その間のターン・アラウンドが極端に低下することとなり，ユーザの使い

*特に過大な数値ではない。



○: MRB off　●: MRB on

図 B-3: MRB

勝手はもちろん，並列プロセッサにおけるプロセッサ間通信にも悪い影響を与える。また，参照の局所性の面からも，単調なメモリ消費は好ましいものではない。

そこで KL1 の処理系では，MRB (Multiple Reference Bit) という一種のリファレンス・カウンタをタグの一部として埋め込み，これを用いて実時間でのガベージ・コレクションを行っている [Chikayam 88, Kimura 90]。MRB はポインタ／オブジェクト共用の1ビットのリファレンス・カウンタ（2以上は全てオーバフロー）と考えることができる。即ち，MRB がオン（多重参照である）というのは；

- 複数のポインタから指されている（可能性がある），または；
- 他のポインタと共に同じセルを指している（可能性がある）；

ことを意味する（図 B-3）。これを換言すると，MRB がオフであるポインタが MRB がオフであるセルを指していれば，他にはポインタが存在しないことが保証される。従って，ポインタの連鎖のルートから終端までの MRB が全てオフであれば，ユニフィケーションなどによってその連鎖を「消費」した時点で，連鎖に属する全てのセルをガベージとして回収することができる。

但しこの規則には例外があり，未定義変数を指しているポインタが2つだけである場合，

この二つのポインタのMRBはオフであって良い*（図B-4）。従って，未定義変数への代入のためのユニフィケーションでは，変数セルへ到達するまでのポインタの連鎖中のMRBが全てオフであっても，変数セルを回収することはできない（ポインタ・セルは回収される）。またこの場合，変数セルに代入する値のMRBが，代入後の変数セルのMRBとなる。逆に，変数セルまでのポインタの中にMRBがオンのものがあった場合，代入後の変数セルのMRBは無条件にオンになる。

さて，セルの回収はユニフィケーション（及び組込述語の実行）の際に行われるが，ガードとボディでは若干回収の方法が異なる。例えば；

```
filter([f(X)|In],FYO):- true | FYO = [f(Y)|Out], modify(X,Y),
filter(In,Out).
```

において，filterの第一引数が単一参照のリスト・セルである時，デレファレンスによってリスト・セルに至るポインタ・セル，即ちReference Pointerとリスト・セルを直接指しているポインタが回収される。一方リスト・セルは，carとf(X)とのユニフィケーションが成功するとは限らないため，コミットされるまでその回収は保留される。また，f(X)に関するユニフィケーションが失敗または中断した時に備えて，他のクローズが回収してしまったポインタを使用することがないように，引数レジスタA_1にデレファレンス結果（リスト・セルを直接指すポインタ）が書戻される。コミットが実行されると，リスト・セルは不要となるため回収が行われるが，これはコンパイラが生成する命令collect_listによって行われる[Kimura 87]（図B-5）。

一方，ボディでのユニフィケーションFYO=[f(Y)|Out]においては，FYOが未定義変数であり†，かつReference PointerのMRBが全てオフである場合には，前述のようにReference Pointerのみが回収される。また，[f(Y)|Out]に対応するリスト・セルが生成され，未定義変数へはそれを指すMRBがオフのポインタが代入される。但しこの例では，[f(X)|In]のリスト・セルが回収されたばかりである可能性が高いので，リスト・セルの回収と生成を同時に行う（即ち再利用する）命令reuse_listが，コンパイラにより生成される[Inamura 89]。また，FYOが単一参照のリスト・セルである場合には，ガードとは異なり，ユニフィケーション時に直ちにリスト・セルが回収される。

回収されたポインタ・セルやリスト・セルなどは，その大きさに応じたフリー・リストに戻される。また，セルの生成時にはフリー・リストの先頭，即ち最近に回収したセルが再利用される。従って，フリー・リストの先頭付近に対するアクセスが支配的になることが期待され，参照の局所性が維持される。

MRBをオンにするのは，コンパイラの指示によって行われる。コンパイラは；

*未定義変数セル自身のMRBは定義されない。
†これが普通である。

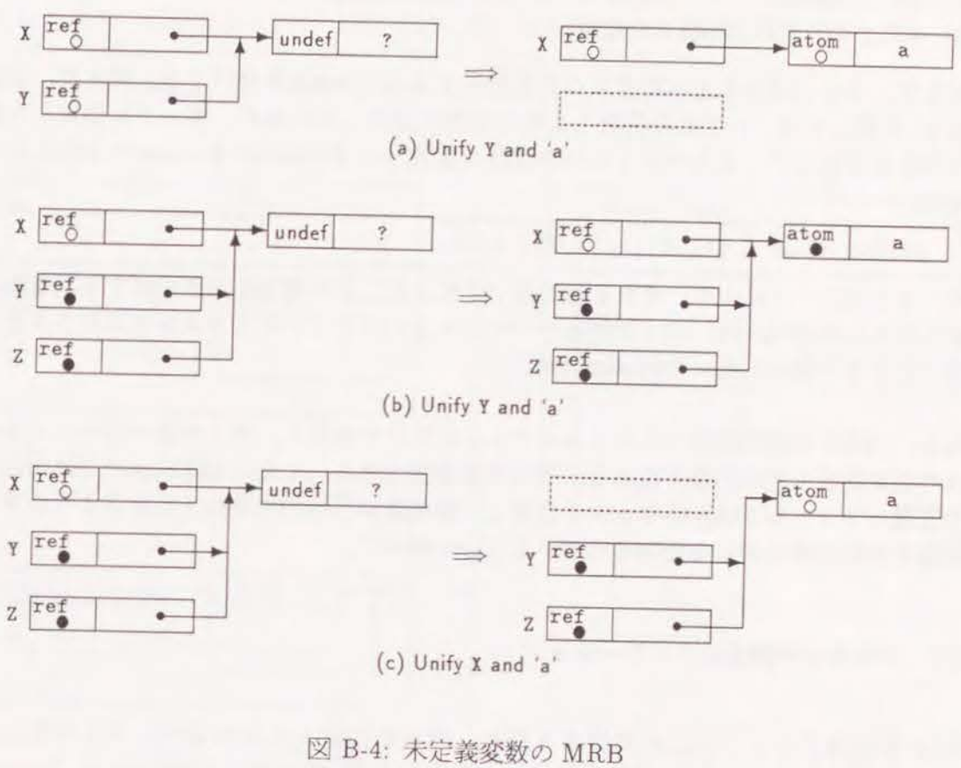


図 B-4: 未定義変数のMRB

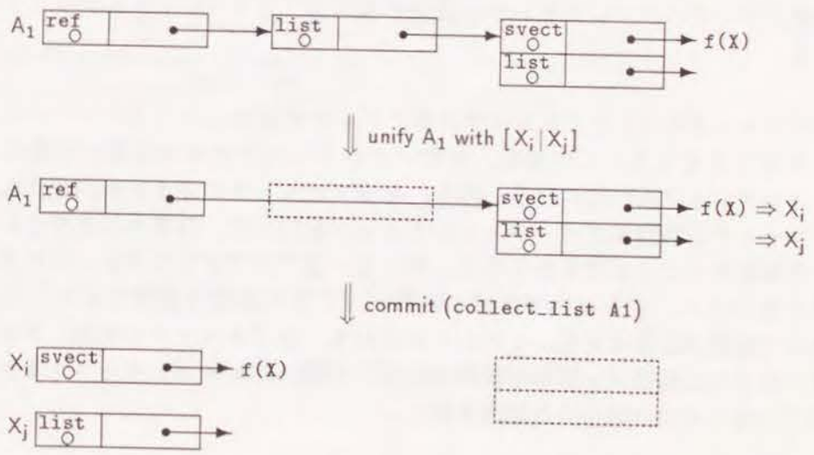


図 B-5: セルの回収

(1) ガードに出現し，かつボディに二回以上出現した変数
(2) ボディに三回以上出現した変数。

に対して，そのMRBをオンにすることを指示する命令markを発行する。例えば，前述のfilterに関しては，いずれの変数も上記の規則に該当しないため，ガードに出現した変数のMRBは変化せず，またボディにのみ現れた変数へのReference PointerのMRBはオフとなる。一方；

```
p(X):- true | q(X,Y), r(X,Y), s(Y).
```

では，Xは規則(1)により，またYは規則(2)により，そのMRBがオンになる。なお，一旦オンになったMRBは，バッチ的なガベージ・コレクションによりオフになりうることが確認されるまで変化しない[Miyauchi 87]。

なお，MRBは実時間ガベージ・コレクションだけではなく，B.1で述べたマージャや構造体の要素更新の効率的な実現にも，不可欠な要素である。また，MRBがオンになった時点で正確なリファレンス・カウンタを生成し，参照数が一時的に2以上になるようなデータの回収を可能とする方法も提案されている[Goto 88a]。

B.2.3 プロセッサ間ユニフィケーション

KL1を並列プロセッサ上に実装する際に，最大の問題となるのはプロセッサ間のユニフィケーションの方式である。特に筆者らが開発したMulti-PSI/v2 [Takeda 88, Uchida 88]や，現在開発中のPIM/m [Nakashima 90c, 90d]のような疎結合のシステムでは，ユニファイすべき変数／データのアドレスをいかに表現するかが，ガベージ・コレクションとの関係で問題となる。

例えば，プロセッサのIDとプロセッサ内のアドレスを結合した，「グローバル」なアドレスを用いることを考える。この場合，変数／データへのアクセス手順は容易であるが，ガベージ・コレクションが困難になる。即ち，あるプロセッサのメモリが消費され尽くした時，そのプロセッサが独自にガベージ・コレクションを行って，外部から参照されている変数／データを移動することができなくなる。従って，全てのプロセッサが一斉にガベージ・コレクションを行うか，または外部参照の変数／データの移動を参照元に対して通知するか，いずれかの処置が必要となる。これらはいずれも，全プロセッサの同期，プロセッサ間を移動している全ての通信メッセージの到着確認，大量のブロード・キャストなど，特に大規模な並列プロセッサでは禁止的な処理を伴う。

そこで，プロセッサ間の参照アドレスを，プロセッサ内のメモリ・アドレスから切り離し，プロセッサ間ユニフィケーションの際に前者から後者へのマッピングを行う方法を採用した[Ichiyoshi 87]。この方法では，図B-6に示すように，外部から参照されている変数／

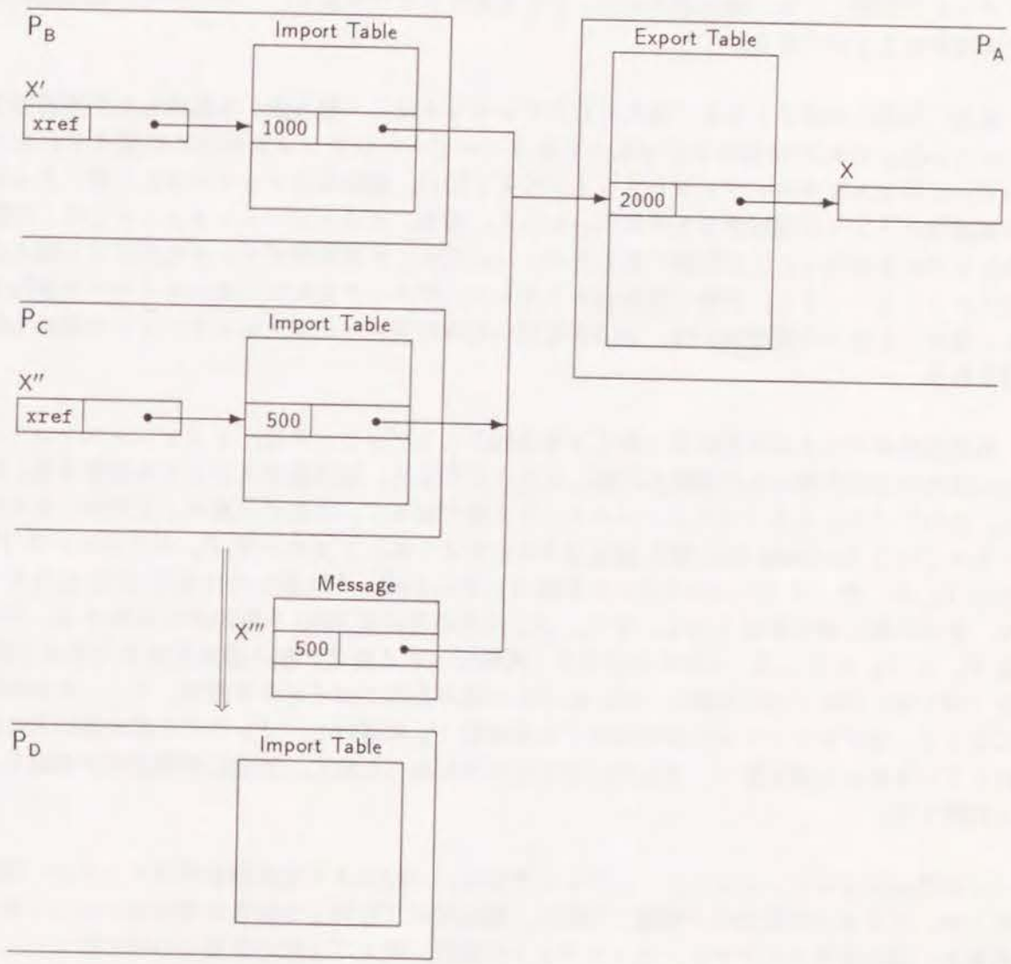


図 B-6: プロセッサ間の参照

データは，「輸出表」という変換テーブルを介してアクセスされる。即ち，他のプロセッサにゴールの実行を依頼する時など，内部の変数／データへのポインタを外部へ「輸出」する際に，そのデータのアドレスを輸出表に登録し，外部へは輸出表のエントリ番号をプロセッサ間アドレスとして公開する。従って，外部から変数／データへのアクセスする際には，輸出表を索引してアドレスを求める必要がある。しかし，ガベージ・コレクションの際に変数／データを移動しても，輸出表のエントリを更新するだけで良く，「ローカル」な処理のみで実現することができる。

また，外部へのポインタを「輸入」したプロセッサは，「輸入表」を経由した間接参照を行っている。これは簡易的なプロセッサ間ガベージ・コレクションのために導入されたもので，ローカルなガベージ・コレクションの完了時に，輸出元のプロセッサに不要となった外部参照ポインタを通知するために用いられる。即ち，ガベージ・コレクションでは「不要な」ものを直接知ることは困難であるため，「必要な」外部参照ポインタに対応する輸入表のエントリをマークし，最後に輸入表をスキャンしてマークされていないポインタを通知する。なお，この「不要通知」は，MRB を用いた実時間ガベージ・コレクションの際にも発行される。

外部参照ポインタが不要になったことを通知されたプロセッサは，リファレンス・カウントにより完全に外部からの参照が消滅したことを知ると，輸出表のエントリを回収する。但し，このリファレンス・カウントはポインタの数ではなく，それに「重み」を付加したものとなっている [Ichiyoshi 88]。例えば図 B-6に示すように，プロセッサ P_A がプロセッサ P_B とと P_C に，データ X へのポインタを輸出している時，それぞれには重み 1000 が与えられ，各々の輸入表に登録される。また，P_A はその合計値 2000 を輸出表に保持する。その後 P_C が P_D に対し X へのポインタを「再輸出」する場合，輸入表に記された重み 1000 を（例えば）500 ずつに分割し，P_D に 500 の重みを持つポインタを渡す。ポインタが不要になると，各プロセッサは自身が保持する重みを P_A に通知し，P_A はその値を輸出表に保持している値から差し引く。そして，この値が 0 になった時に，完全に外部参照が消滅したと判断する。

この重み付きのリファレンス・カウント方式は，上記のような外部参照ポインタの「重複化」や，プロセッサ間での「移動」の際に，輸出元に「許可」を得る必要がないという利点がある。即ち通常のリファレンス・カウントの場合，例えば上記の重複化の例では；

(1) P_C から P_A にポインタ増加を通知する。
(2) P_A はリファレンス・カウンタに 1 を加え，その旨を P_C に返答する。
(3) P_C は返答を受取った後に P_D にポインタを輸出する。
(4) P_D が既にそのポインタを輸入していれば，P_A にポインタ減少を通知する。

という処理が必要である*。一方重み付きの場合，(1), (2), (4) が全て不要であるため，プロセッサ間通信量の削減と，ポインタ移動の時間短縮の，二重のメリットがある。

なお，同様のリファレンス・カウント方法が，一つの荘園に属するゴールが複数のプロセッサで実行される場合の，荘園全体の実行完了判定や強制終了の処理にも用いられている [Rokusawa 88]。

*(2), (3) を省くとポインタ増加通知が，後に行われる可能性のある P_D からのポインタ減少通知に追い越される危険がある。

付録 C　WAM 命令一覧

本項では [Warren 83] に示された WAM 命令の機能と，PSI-II, PSI-III における命令実行サイクル数を示す。なお実現方式の違いにより [Warren 83] とは一部異っており，削除されたもの（get_structure など），追加されたもの（get_atom など），機能が変更されたもの（allocate など）がある。

機能の表記は C のシンタックスに準拠し，以下の記法を用いる。

A[i], X[i]	引数／一時変数レジスタ
P	プログラム・カウンタ
CP	復帰アドレス
E	最新の Environment のベース。Environment の要素は以下のものである。 E->old_E　直前の Environment へのポインタ E->CP　復帰アドレス E->Y[n]　局所変数
B	最新の Choice Point のベース。Choice Point の要素は以下のものである。 B->old_B　直前の Choice Point へのポインタ B->E　Environment へのポインタ B->CP　復帰アドレス B->G　グローバル・スタック・トップ B->TR　トレイル・スタック・トップ B->AP　候補節のアドレス B->N　引数の数 B->A[i]　引数レジスタ
L	ローカル・スタック・トップ
G	グローバル・スタック・トップ
GB	グローバル・スタック・トップのバックトラック・ポイント
TR	トレイル・スタック・トップ
S	構造体ポインタ
MODE	構造体のユニフィケーションが読出モード (read) か書込モード (write) かを示す。

X.tag	X のタグ部
X.value	X の値部
ref	Reference Pointer のタグ
undef	未定義変数のタグ
atom	アトムのタグ
int	整数のタグ
list	リストのタグ
vect	ベクタのタグ
unify(X,Y)	X と Y とのユニフィケーションを行う。
data(T,V)	タグ部が T，値部が X のデータを返す。
vect_data(N,A)	要素数が N であるようなベクタを指示するポインタを返す。
length(V)	ベクタ V の要素数を返す。
deref(X)	X のデレファレンス結果を，以下のように求める。 arg = X ; while (arg.tag == ref) arg = *(arg.value) ; return(arg) ;
local_var(X)	X が局所変数であれば真，大域変数であれば偽を返す。

bind(X,D)	未定義変数 X へのデータ D の代入を，以下のように行う。
	*(X.value) = D ;
	if (local_var(X)) {
	if (X.value < B)
	*TR++ = X ; }
	else {
	if (X.value < GB)
	*TR++ = X ; }

jump(L)	L へ分岐する。

fail()	バックトラック処理を以下のように行う。
	E = B->E ; CP = B->CP ; G = B->G ;
	old_TR = TR ; TR = B->TR ; L = B + B->N + 6 ;
	for (i = 1 ; i <= B->N ; i++)
	A[i] = B->A[i] ;
	while (--old_TR >= TR)
	*(*old_TR) = *old_TR ;
	jump(B->AP) ;

hash_for_const(C)	定数 C のハッシュ値を求める。
hash_for_vect(V)	ベクタ V の要素数と第一要素によりハッシュ値を求める。

命令の実行サイクル数はオペランドによって異り，特にユニフィケーションを行う命令はデレファレンスの回数，成功か失敗か，未定義変数の代入の際のトレイルの要否などにより大きく異る。そこで；

$\langle case \rangle \rightarrow \langle cycle \rangle$

のように，代表的な $\langle case \rangle$ におけるサイクル数 $\langle cycle \rangle$ を表示する。なお，$\langle case \rangle$ の意味は以下の通りである。

`d=d` : `get_value` などにおいて，双方のデータがデレファレンス不要なアトミック・データであり，かつユニフィケーションに成功した場合。

`d=V` : `get_value` などにおいて，一方のデータがデレファレンス不要なアトミック・データであり，他方が一回のデレファレンスで到達できるトレイル不要の未定義変数である場合。

`d` : `get_constant` などにおいて，ユニフィケーション対象がデレファレンス不要なデータであり，かつユニフィケーションに成功した場合。

`V` : `get_constant` などにおいて，ユニフィケーション対象が一回のデレファレンスで到達できるトレイル不要の未定義変数である場合。

`r` : *unify* 系の命令が読出モードで実行された場合。

`w` : *unify* 系の命令が書込モードで実行された場合。

C.1 get 系命令

命令とその機能	サイクル数 (PSI-II)	サイクル数 (PSI-III)
`get_variable Xn,Ai` `  X[n] = A[i] ;`	1	1
`get_variable Xn,Ai` `  E->Y[n] = A[i] ;`	3	1
`get_value Xn,Ai` `  unify(X[n], A[i]) ;`	`d=d→4` `d=V→6`	`d=d→2` `d=V→4`
`get_value Yn,Ai` `  unify(E->Y[n], A[i]) ;`	`d=d→5` `d=V→7`	`d=d→2` `d=V→4`
`get_constant C,Ai` `  unify(C, A[i]) ;`	`d→3` `V→5`	`d→4` `V→5`
`get_atom C,Ai` `  unify(data(atom, C), A[i]) ;`	`d→2` `V→6`	`d→2` `V→3`
`get_integer C,Ai` `  unify(data(int, C), A[i]) ;`	`d→2` `V→6`	`d→2` `V→3`

`get_list Ai`
`  arg = deref(A[i]) ;`
`  switch (arg.tag) {`
`  case list : S = arg.value ; MODE = read ;`
`  break ;`
`  case undef : bind(arg, data(list, G)) ;`
`  MODE = write ; break ;`
`  default : fail() ; break ;}`
	`d→1`
	`V→5`
		`d→1`
		`V→3`

`get_vector N,Ai`
`  arg = deref(A[i]) ;`
`  switch (arg.tag) {`
`  case vect :`
`  if (length(arg) == N) {`
`  S = arg.value ; MODE = read ; break ; }`
`  else {`
`  fail() ; break ; }`
`  case undef : bind(arg, vect_data(N, G)) ;`
`  MODE = write ; break ;`
`  default : fail() ; break ;}`
	`d→2`
	`V→4`
		`d→1`
		`V→3`

C.2 put 系命令

命令とその機能	サイクル数 (PSI-II)	サイクル数 (PSI-III)
put_variable Xn,Ai `X[n] = A[i] = data(ref, G) ; *G = data(undef, G++) ;`	3	3
put_variable Yn,Ai `A[i] = data(ref, &(E->Y[n])) ;`	1	1
put_value Xn,Ai `A[i] = X[n] ;`	1	1
put_value Yn,Ai `A[i] = E->Y[n] ;`	1	1

put_unsafe_value Yn,Ai
`arg = deref(E->Y[n]) ;`
`if (arg.tag == undef && arg.value >= E) {`
`  A[i] = data(ref, G) ; *G = data(undef, G++) ; }`
`else`
`  A[i] = arg ;`
	4	2

put_constant C,Ai `A[i] = C ;`	2	2
put_atom C,Ai `A[i] = data(atom, C) ;`	1	1
put_integer C,Ai `A[i] = data(int, C) ;`	1	1
put_list Ai `A[i] = data(list, G) ; MODE = write ;`	1	1
put_vector N,Ai `A[i] = vect_data(N, G) ; MODE = write ;`	1	1

C.3 unify 系命令

命令とその機能	サイクル数 (PSI-II)	サイクル数 (PSI-III)

unify_variable Xn
`if (MODE == read)`
`  X[n] = data(ref, S++) ;`
`else {`
`  X[n] = data(ref, G) ; *G = data(undef, G++) ; }`
	r→1
	w→2
		r→1
		w→1

命令とその機能	サイクル数 (PSI-II)	サイクル数 (PSI-III)

unify_variable Yn
`if (MODE == read)`
`  E->Y[n] = data(ref, S++) ;`
`else {`
`  E->Y[n] = data(ref, G) ; *G = data(undef, G++) ; }`
	r→3
	w→4
		r→2
		w→2

unify_value Xn
`if (MODE == read)`
`  unify(X[n], *S++) ;`
`else {`
`  arg = deref(X[n]) ;`
`  if (arg.tag == undef && local_var(arg)) {`
`  *arg = data(ref, G) ; *G++ = data(undef, G++) ; }`
`  else`
`  *G++ = arg ; }`
	r,d=d→5
	r,d=V→7
	w→2
		r,d=d→3
		r,d=V→5
		w→1

unify_value Yn
`if (MODE == read)`
`  unify(E->Y[n], *S++) ;`
`else {`
`  arg = deref(E->X[n]) ;`
`  if (local_var(arg)) {`
`  *arg = data(ref, G) ; *G++ = data(undef, G++) ; }`
`  else`
`  *G++ = arg ; }`
	r,d=d→6
	r,d=V→8
	w→3
		r,d=d→3
		r,d=V→5
		w→1

unify_constant C
`if (MODE == read)`
`  unify(C, *S++) ;`
`else`
`  *G++ = C ;`
	r,d→5
	r,V→5
	w→3
		r,d→6
		r,V→5
		w→2

unify_atom C
`if (MODE == read)`
`  unify(data(atom, C), *S++) ;`
`else`
`  *G++ = data(atom, C) ;`
	r,d→5
	r,V→5
	w→3
		r,d→4
		r,V→5
		w→1

unify_integer C
`if (MODE == read)`
`  unify(data(int, C), *S++) ;`
`else`
`  *G++ = data(int, C) ;`
	r,d→5
	r,V→5
	w→3
		r,d→4
		r,V→5
		w→1

C.4 control 系命令

命令とその機能	サイクル数 (PSI-II)	サイクル数 (PSI-III)
call Lab CP = P + 1 ; jump(Lab) ;	4	3
execute Lab jump(Lab) ;	3	2
proceed jump(CP) ;	4	4
allocate N old_E = E ; E = L ; E->old_E = old_E ; E->CP = CP ; L = L + N + 2 ; for (i = 1 ; i <= N ; i++) E->Y[i] = data(undef, &(E->Y[i])) ;	8+N	5+N
deallocate E = E->old_E ; if (E > B) L = E ;	E<B→4 E>B→5	E<B→4 E>B→6

C.5 indexing 系命令

命令とその機能	サイクル数 (PSI-II)	サイクル数 (PSI-III)
try_me_else Lab,N old_B = B ; B = L ; L = L + N + 6 ; B->old_B = old_B ; B->E = E ; B->CP = CP ; B->G = GB = G ; B->TR = TR ; B->AP = Lab ; B->N = N ; for (i = 1 ; i <= N ; i++) B->A[i] = A[i] ;	11+N	9+N
try Lab,N old_B = B ; B = L ; L = L + N + 6 ; B->old_B = old_B ; B->E = E ; B->CP = CP ; B->G = GB = G ; B->TR = TR ; B->AP = P + 1 ; B->N = N ; for (i = 1 ; i <= N ; i++) B->A[i] = A[i] ; jump(Lab) ;	12+N	9+N
retry_me_else Lab B->AP = Lab ;	3	2
retry Lab B->AP = P + 1 ; jump(Lab) ;	5	2

命令とその機能	サイクル数 (PSI-II)	サイクル数 (PSI-III)
trust_me L = B ; B = B->old_B ; GB = B->G ;	5	5
trust Lab L = B ; B = B->old_B ; GB = B->G ; jump(Lab) ;	5	5
switch_on_term Ai,Lv,Lc,Ll,Ls arg = deref(A[i]) ; switch (arg.tag) { case atom : case int : jump(Lc) ; break ; case list : jump(Ll) ; break ; case vect : jump(Ls) ; break ; default : jump(Lv) ; break ; }	Lv→6 Lc→4 Ll→5 Ls→5	Lv→6 Lc→3 Ll→6 Ls→4
switch_on_constant Ai,Table jump(Table[hash_for_const(A[i])]) ;	9	12
switch_on_structure Ai,Table jump(Table[hash_for_vect(A[i])]) ;	15	20

付録 D　WAM の実行例

本項では 2.1 に示したプログラム例を用いて，対応する WAM のコードと実行過程でのスタックなどの状態を示す。プログラムは以下のものである*。

```
elder_brother(X, Y):- brother(X, Y), elder(X, Y).
brother(X, Y):- father(F, X), father(F, Y).

father(家康, 信康).
father(家康, 秀忠).
father(家康, 秀康).

elder(信康, 秀忠).
elder(信康 , 秀康).
elder(秀忠 , 秀康).
```

上記のプログラムをコンパイルすると，図 D-1のような WAM のコードが生成される。また，プログラムをゴール；

```
elder_brother(秀忠, B)
```

により呼び出すと，実行が成功裏に完了し，未定義変数 B に値秀康が代入される。以下，この実行過程を順を追って説明する。

*Environment のネスティングなどを示すため elder_brother の定義を一部修正している


```
elder_brother:
[ 1]  allocate          2           % elder_brother(
[ 2]  get_variable      Y1, A1      %   X,
[ 3]  get_variable      Y2, A2      %   Y) :-
[ 4]  call              brother     % brother(X,Y),
[ 5]  put_value         Y1, A1      % elder(X,
[ 6]  put_value         Y2, A2      %   Y).
[ 7]  deallocate                    %
[ 8]  execute           elder       %
brother:
[ 9]  allocate          2           % brother(X,
[10]  get_variable      Y1, A2      %   Y):-
[11]  put_value         A1, A2      % father(
[12]  put_variable      Y2, A1      %   F, X),
[13]  call              father      %
[14]  put_unsafe_value  Y2, A1      % father(F,
[15]  put_value         Y1, A2      %   Y).
[16]  deallocate                    %
[17]  execute           father      %

father:
[18]  switch_on_term    father_c1, father_c1, fail, fail
father_c1:
[19]  try_me_else       father_c2   % father(
[20]  get_constant      家康 , A1    %   家康 ,
[21]  get_constant      信康 , A2    %   信康).
[22]  proceed                       %
father_c2:
[23]  retry_me_else     father_c3   % father(
[24]  get_constant      家康 , A1    %   家康 ,
[25]  get_constant      秀忠 , A2    %   秀忠).
[26]  proceed                       %
father_c3:
[27]  trust_me                      % father(
[28]  get_constant      家康 , A1    %   家康 ,
[29]  get_constant      秀康 , A2    %   秀康).
[30]  proceed                       %
```

図 D-1: elder_brother, brother, father, elder のコンパイル・コード (1/2)

```
elder:
[31]  switch_on_term    elder_c1, elder_idx, fail, fail
elder_idx:
[32]  switch_on_constant    <信康:elder_idx1>, <秀忠:elder_c3a>
elder_idx1:
[33]  try               elder_c1a
[34]  trust             elder_c2a
elder_c1:
[35]  try_me_else       elder_c2
                                    % elder(
elder_c1a:                          %   信康,
[36]  get_constant      信康, A1    %   秀忠).
[37]  get_constant      秀忠, A2    %
[38]  proceed                       %
elder_c2:
[39]  retry_me_else     elder_c3
                                    % elder(
elder_c2a:                          %   信康,
[40]  get_constant      信康, A1    %   秀康).
[41]  get_constant      秀康, A2    %
[42]  proceed                       %
elder_c3:
[43]  trust_me
                                    % elder(
elder_c3a:                          %   秀忠,
[44]  get_constant      秀忠, A1    %   秀康).
[45]  get_constant      秀康, A2    %
[46]  proceed                       %
```

図 D-1: elder_brother, brother, father, elder のコンパイル・コード (2/2)

(1) 初期状態

初期状態，即ち elder_brother が呼び出される直前のローカル・スタック，トレイル・スタック，及びレジスタ $\mathbf{P}, \mathbf{CP}, \mathbf{E}, \mathbf{B}, \mathbf{L}, \mathbf{TR}, \mathbf{A_1}, \mathbf{A_2}$ 状態を図 D-2に示す。なお，グローバル・スタック及び関連するレジスタの状態は一切変化しないため，図示していない。

ポインタの表記 " ⇒ [x]" は，図 D-1の命令 [x] を指示していることを意味する。また，表記 " ⇒ <+x>"，"(x)" はローカル・スタック，トレイル・スタックのアドレス <+x>，(+x) を指示していることをそれぞれ意味する。更に，"[?]" はコード領域の任意の場所を，"<?>" はローカル・スタックの任意の場所を，それぞれ意味する。

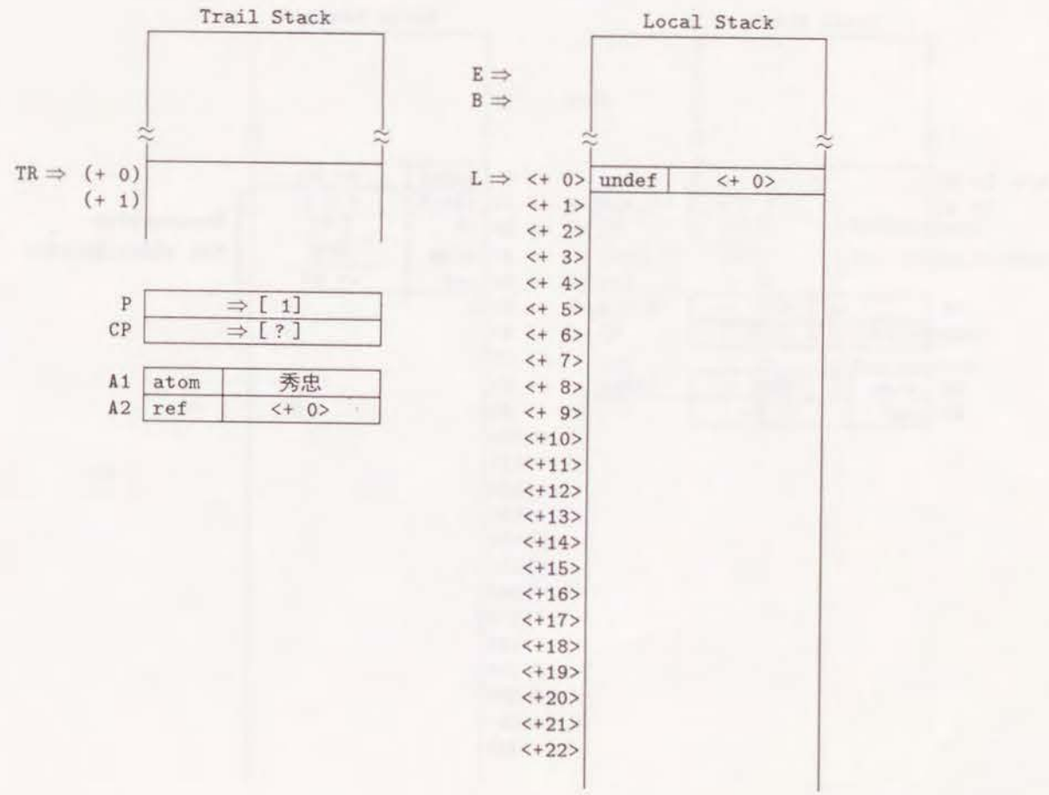


図 D-2: 初期状態

(2) brother の呼び出し

elder_brother が呼び出されると，まず以下の命令列が実行され，brother が呼び出される。

```
[ 1]  allocate          2          %
[ 2]  get_variable      Y1, A1     % A1: 秀忠
[ 3]  get_variable      Y2, A2     % A2:unbound(B)
[ 4]  call              brother    %
```

図 D-3に，この時点での状態を示す。

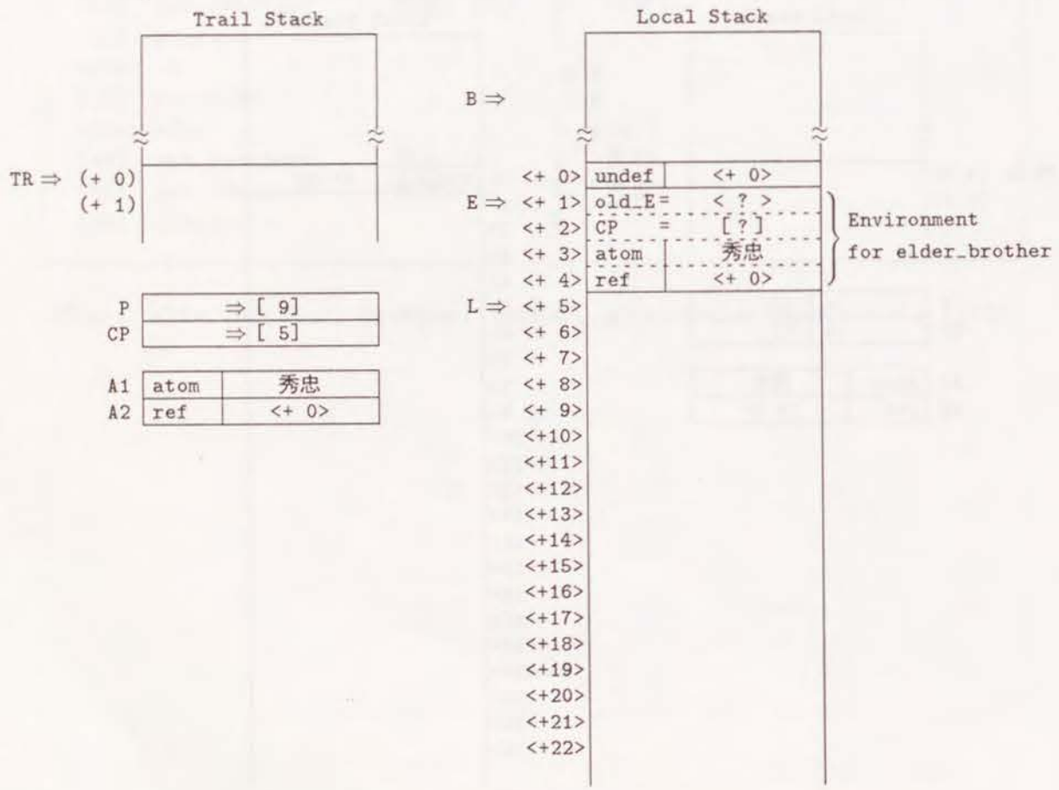


図 D-3: brother の呼び出し

(3) father の呼び出し — 1

次に，以下に示す述語 brother の命令列が実行され，father が呼び出される。

```
[ 9]  allocate          2          %
[10]  get_variable      Y1, A2     % A2=unbound(B)
[11]  put_value         A1, A2     % A2= 秀忠
[12]  put_variable      Y2, A1     % A1=unbound(F)
[13]  call              father     %
```

図 D-4に，この時点での状態を示す。

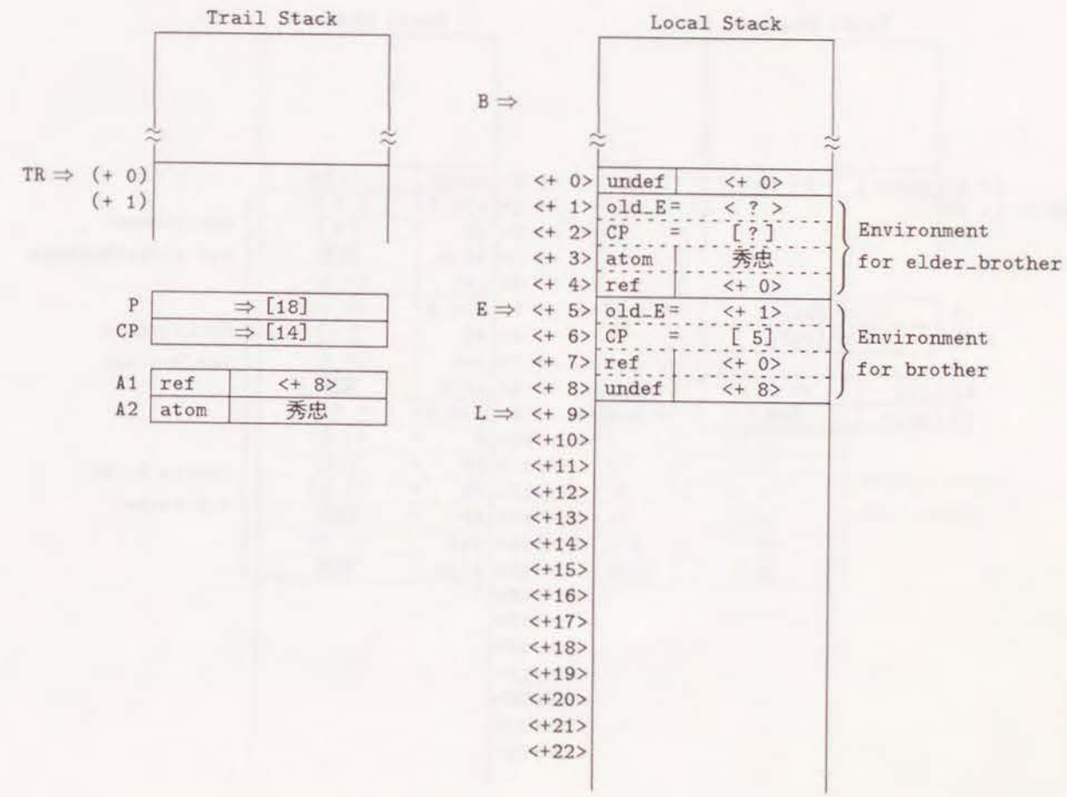


図 D-4: father の呼び出し — 1

(4) father(家康，信康) の実行 — 1

次に，以下に示す述語 father の第一クローズの命令列が実行され，第一引数のユニフィケーションが行われる。

```
[18]  switch_on_term    father_c1, father_c1, fail, fail
[19]  try_me_else       father_c2   %
[20]  get_constant      家康，A1    % A1=unbound(F)-> 家康
```

図 D-5に，この時点での状態を示す。

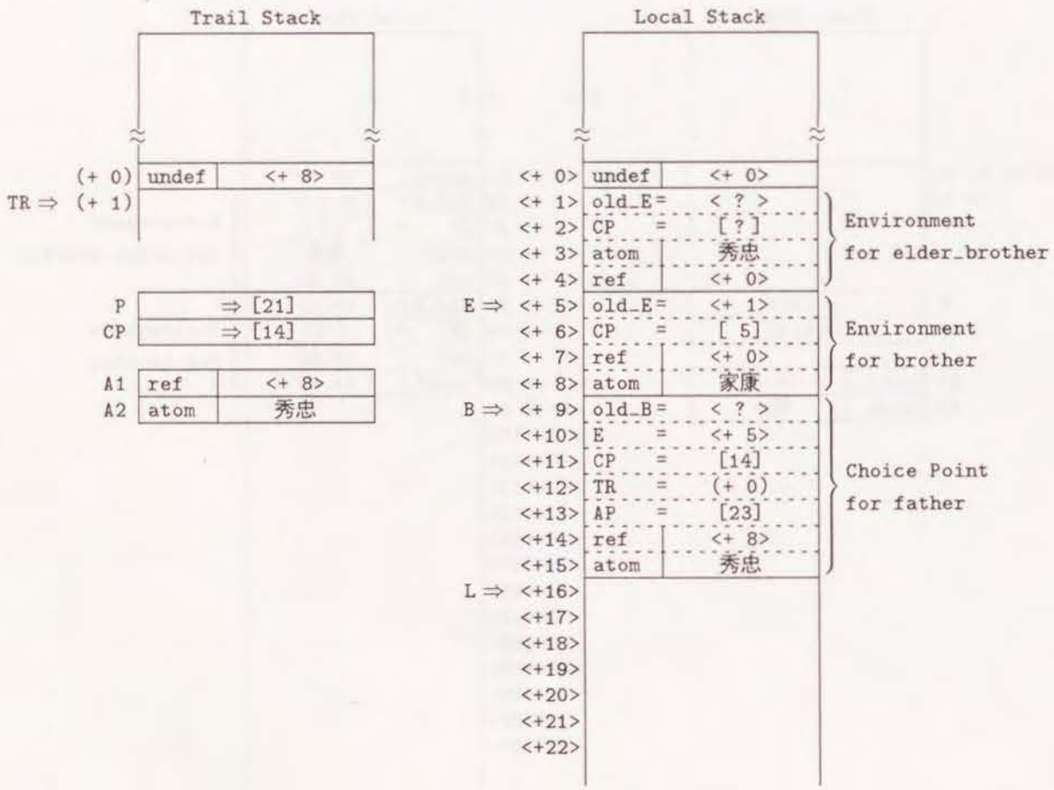


図 D-5: father(家康，信康) の実行 — 1

(5) father(家康，信康) の失敗

次に，第二引数のユニフィケーションが行われるが，これは秀忠と信康のユニフィケーションであるため失敗し，第二クローズへ移行する。即ち；

```
[21]  get_constant      信康，A2    % A2= 秀忠 ->fail
```

が実行される。図 D-6に，この時点での状態を示す。

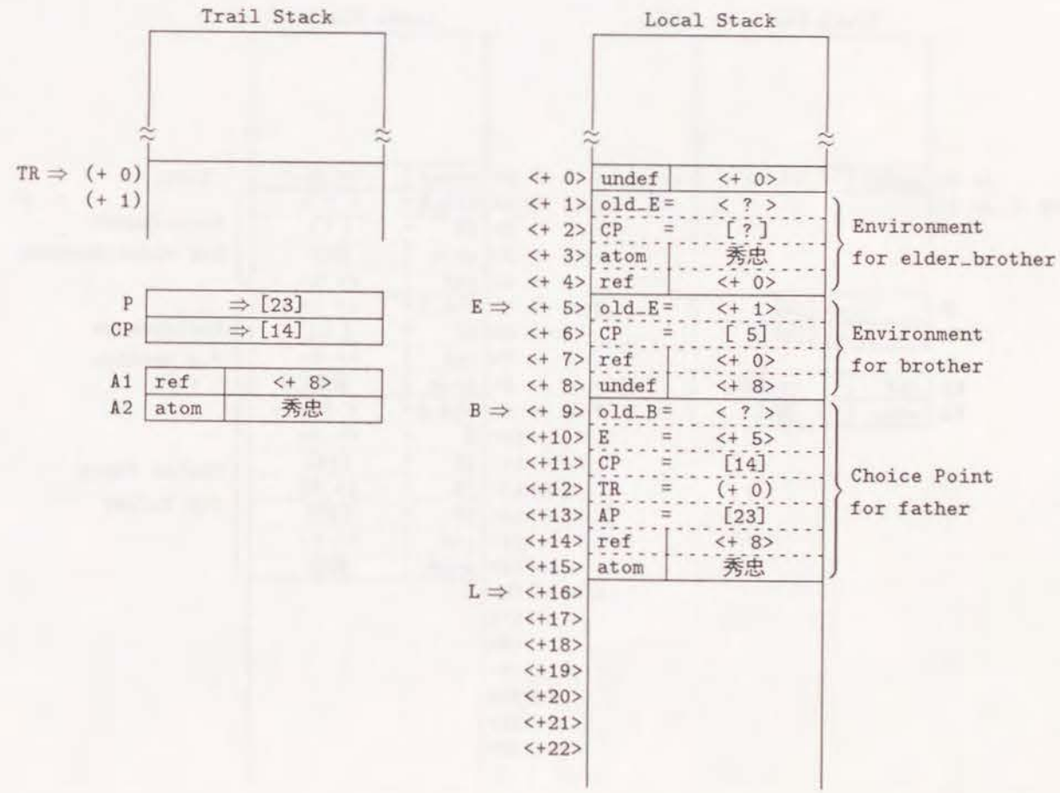


図 D-6: father(家康，信康) の失敗

(6) father(家康，秀忠) の実行 — 1

次に，以下に示す述語 father の第二クローズの命令列が実行され，ユニフィケーションがいずれも成功して brother に戻る。

```
[23]  retry_me_else     father_c3   %
[24]  get_constant      家康，A1    % A1=unbound(F)-> 家康
[25]  get_constant      秀忠，A2    % A2= 秀忠
[26]  proceed                       %
```

図 D-7に，この時点での状態を示す。

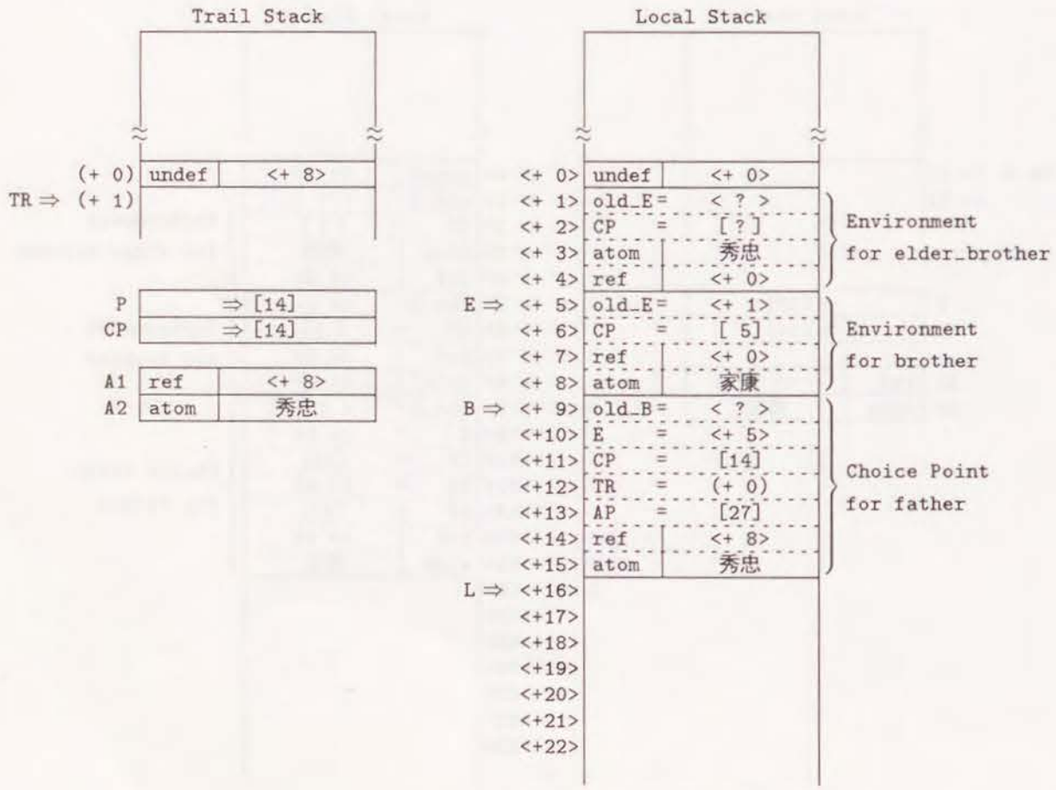


図 D-7: father(家康，秀忠) の実行 — 1

(7) father の呼び出し — 2

次に，以下に示す述語 brother の命令列が実行され，father が再び呼び出される。

```
[14]  put_unsafe_value  Y2, A1      % A1: 家康
[15]  put_value         Y1, A2      % A2:unbound(B)
[16]  deallocate                    %
[17]  execute           father      %
```

図 D-8に，この時点での状態を示す。

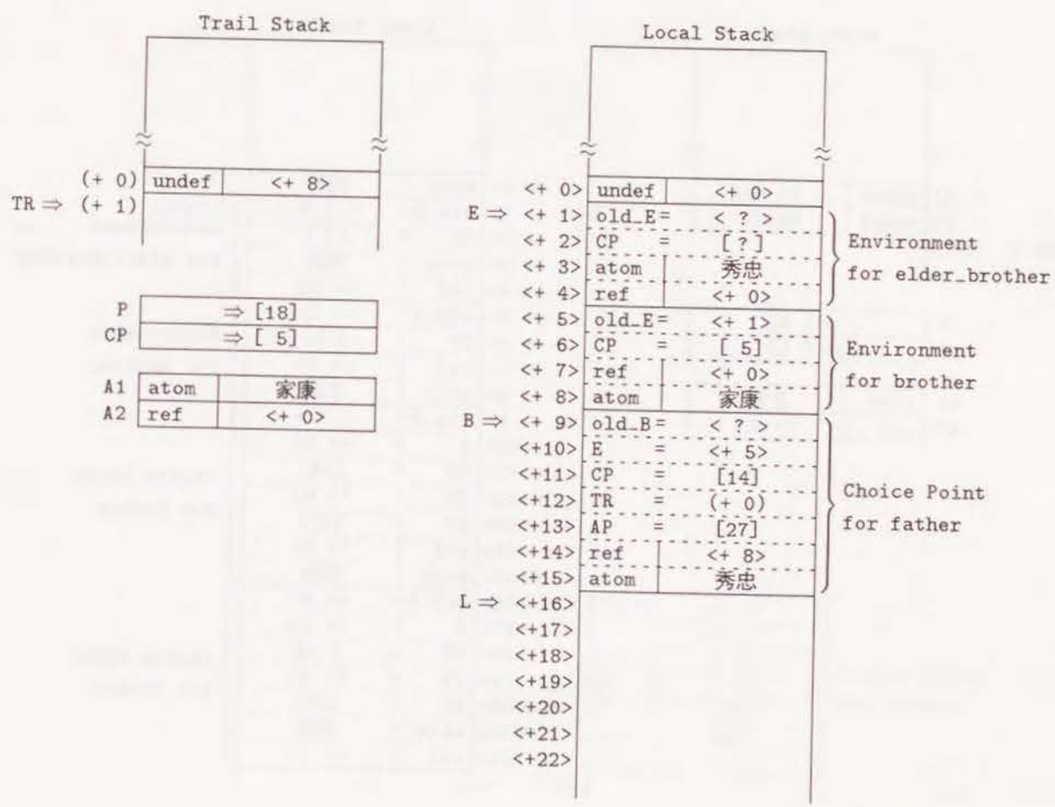


図 D-8: father の呼び出し — 2

(8) father(家康，信康) の実行 — 2

次に，以下に示す述語 father の第一クローズの命令列が実行され，ユニフィケーションがいずれも成功して elder_brother に戻る。

```
[18]  switch_on_term    father_c1, father_c1, fail, fail
[19]  try_me_else       father_c2   %
[20]  get_constant      家康，A1    % A1=家康
[21]  get_constant      信康，A2    % A2=unbound(B)-> 信康
[22]  proceed                       %
```

図 D-9に，この時点での状態を示す。

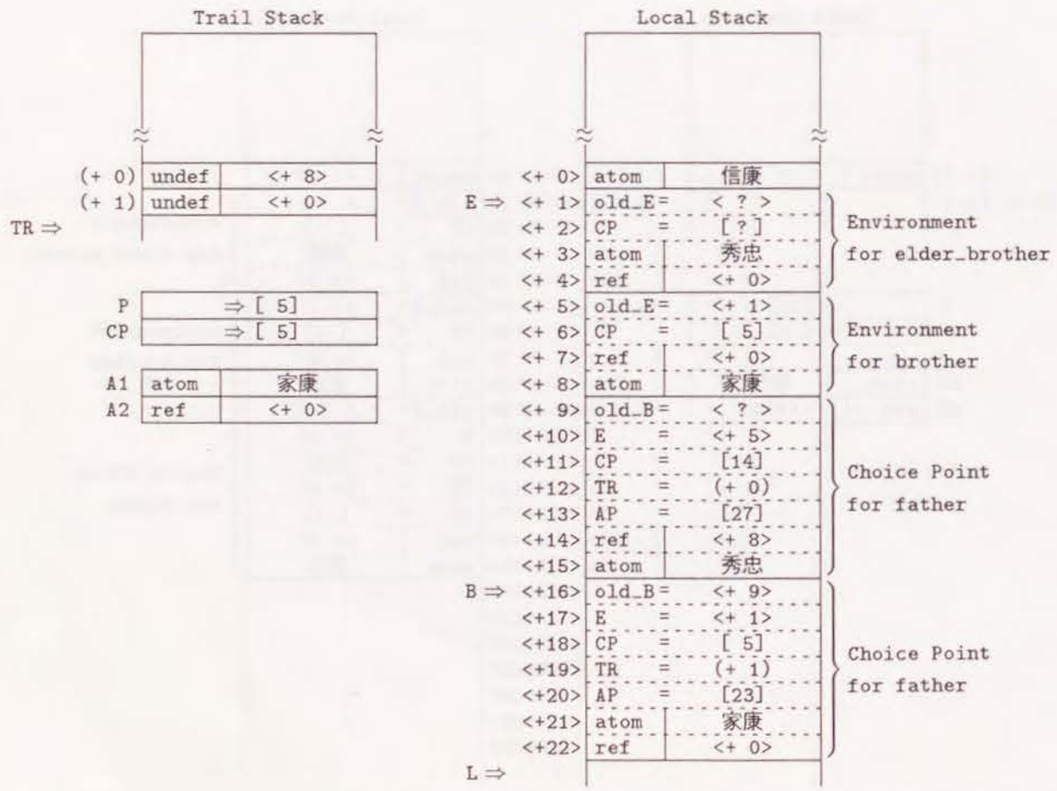


図 D-9: father(家康，信康) の実行 — 2

(9) elder の呼び出し — 1

次に，以下に示す述語 elder_brother の命令列が実行され，elder が呼び出される。

```
[ 5]  put_value         Y1, A1      % A1=秀忠
[ 6]  put_value         Y2, A2      % A2=ref(B)=信康
[ 7]  deallocate                    %
[ 8]  execute           elder       %
```

図 D-10に，この時点での状態を示す。

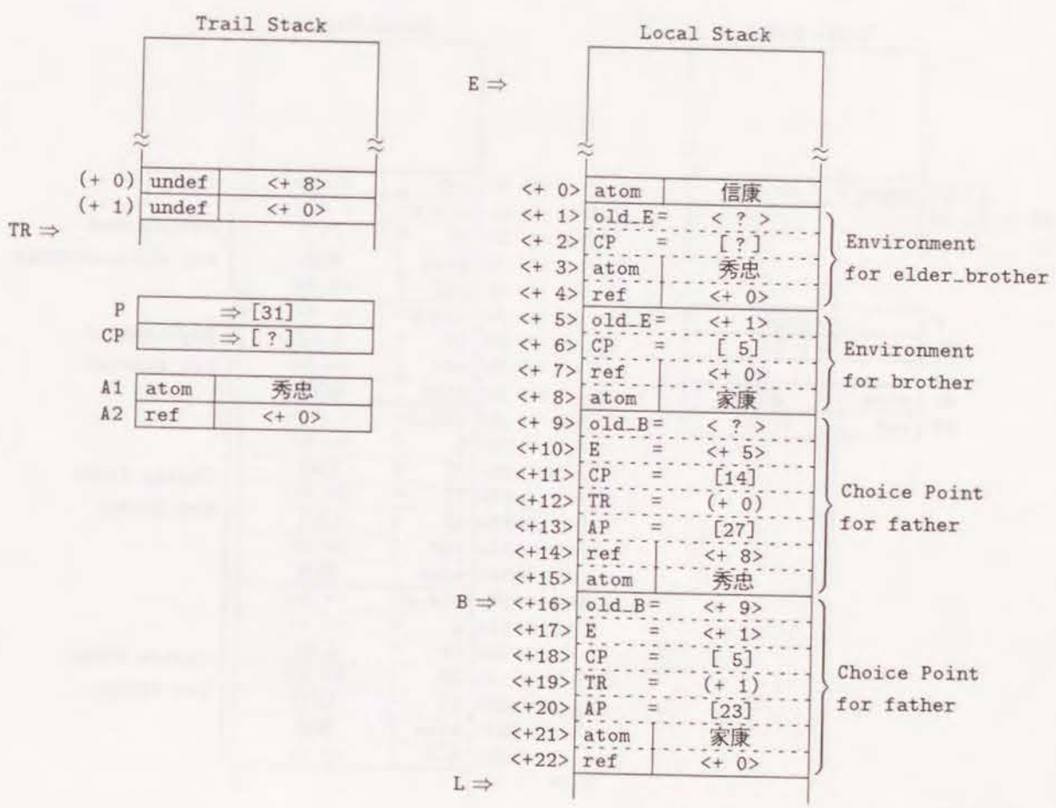


図 D-10: elder の呼び出し — 1

(10) elder の失敗 — 1

次に，以下に示す述語 elder の命令列が実行されるが，第二引数のユニフィケーション，即ち信康と秀康のユニフィケーションに失敗し，father の第二クローズに移行する。

```
[31]  switch_on_term     elder_c1, elder_idx, fail, fail
[32]  switch_on_constant     <信康:elder_idx1>, <秀忠:elder_c3a>
[44]  get_constant       秀忠, A1      % A1= 秀忠
[45]  get_constant       秀康, A2      % A2= 信康->fail
```

図 D-11に，この時点での状態を示す。

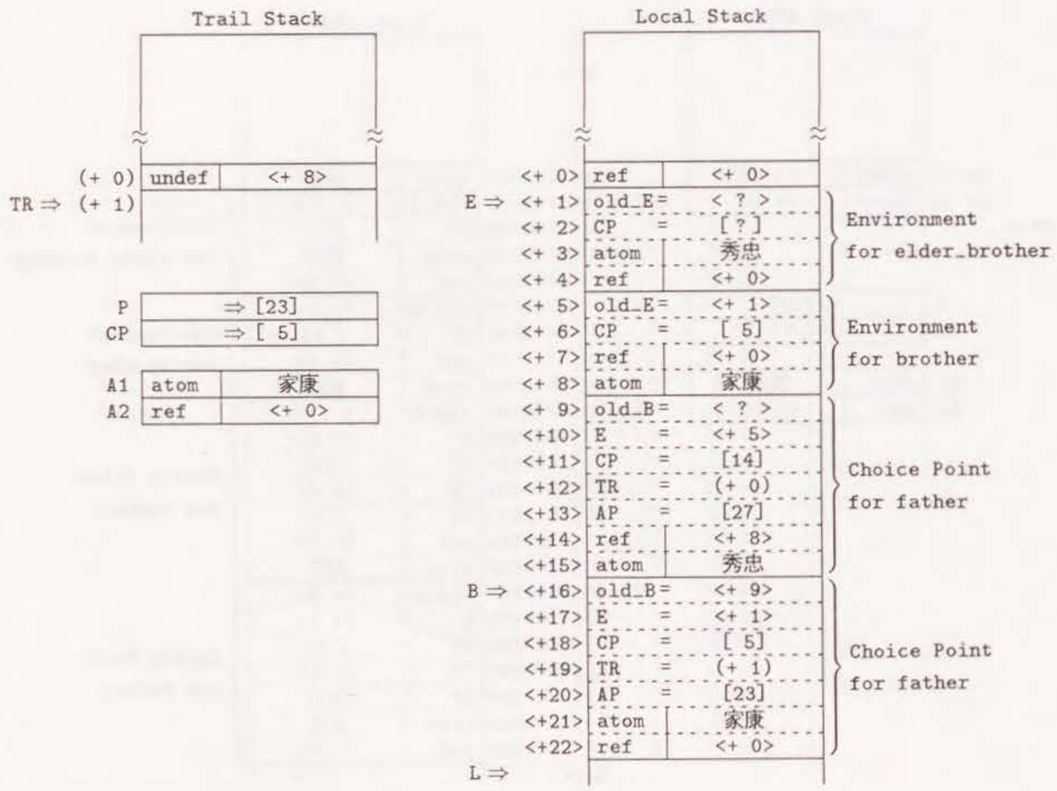


図 D-11: elder の失敗 — 1

(11) father(家康，秀忠) の実行 — 2

次に，以下に示す述語 father の第二クローズの命令列が実行され，ユニフィケーションがいずれも成功して elder_brother に戻る。

```
[23]  retry_me_else      father_c3     %
[24]  get_constant       家康, A1      % A1= 家康,
[25]  get_constant       秀忠, A2      % A2=unbound(B)-> 秀忠
[26]  proceed                          %
```

図 D-12に，この時点での状態を示す。

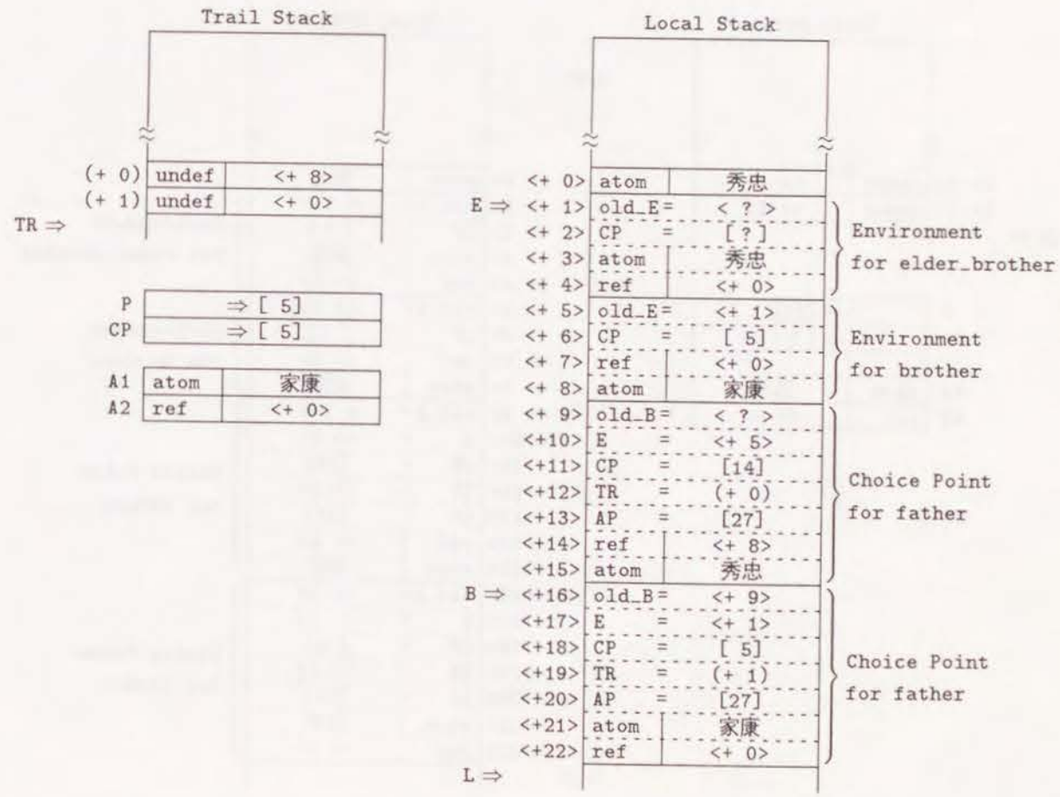


図 D-12: father(家康，秀忠) の実行 — 2

(12) elder の呼び出し — 2

次に，以下に示す述語 elder_brother の命令列が実行され，elder が再び呼び出される。

```
[ 5]  put_value        Y1, A1     % A1=秀忠
[ 6]  put_value        Y2, A2     % A2=ref(B)=秀忠
[ 7]  deallocate                  %
[ 8]  execute          elder      %
```

図 D-13に，この時点での状態を示す。

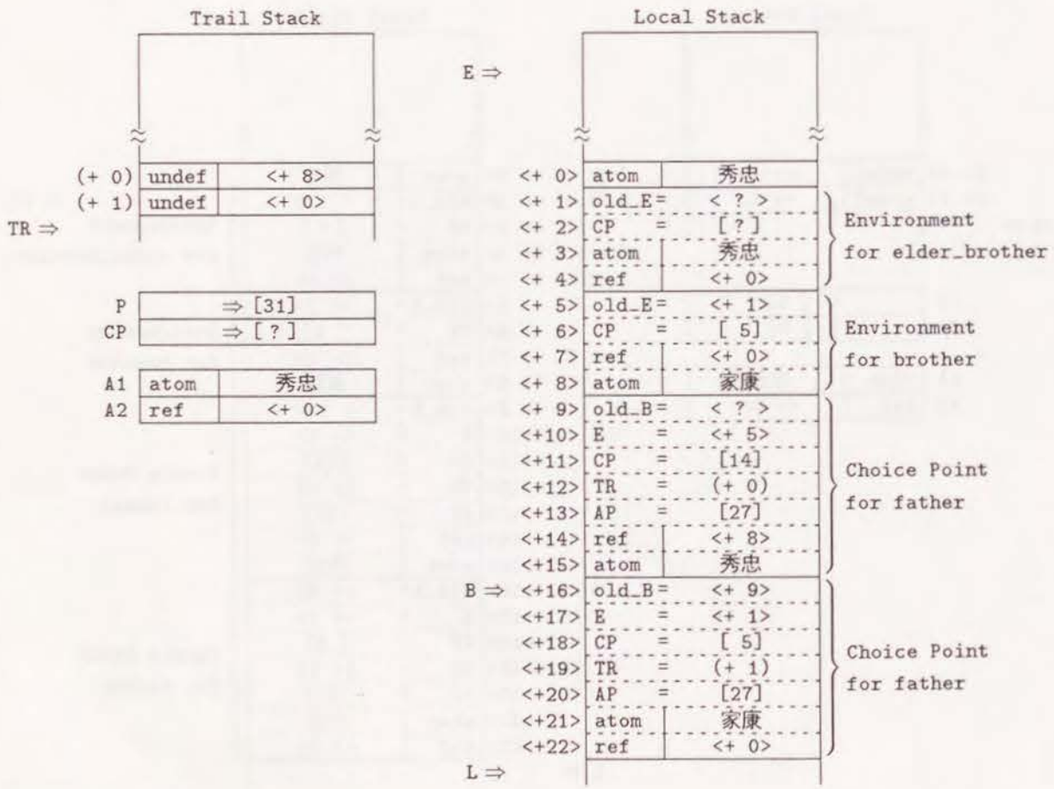


図 D-13: elder の呼び出し — 2

(13) elder の失敗 — 2

次に，以下に示す述語 elder の命令列が実行されるが，第二引数のユニフィケーション，即ち秀忠と秀康のユニフィケーションに失敗し，father の第三クローズに移行する。

```
[31]  switch_on_term     elder_c1, elder_idx, fail, fail
[32]  switch_on_constant     <信康:elder_idx1>, <秀忠:elder_c3a>
[44]  get_constant       秀忠, A1     % A1=秀忠
[45]  get_constant       秀康, A2     % A2=秀忠->fail
```

図 D-14に，この時点での状態を示す。

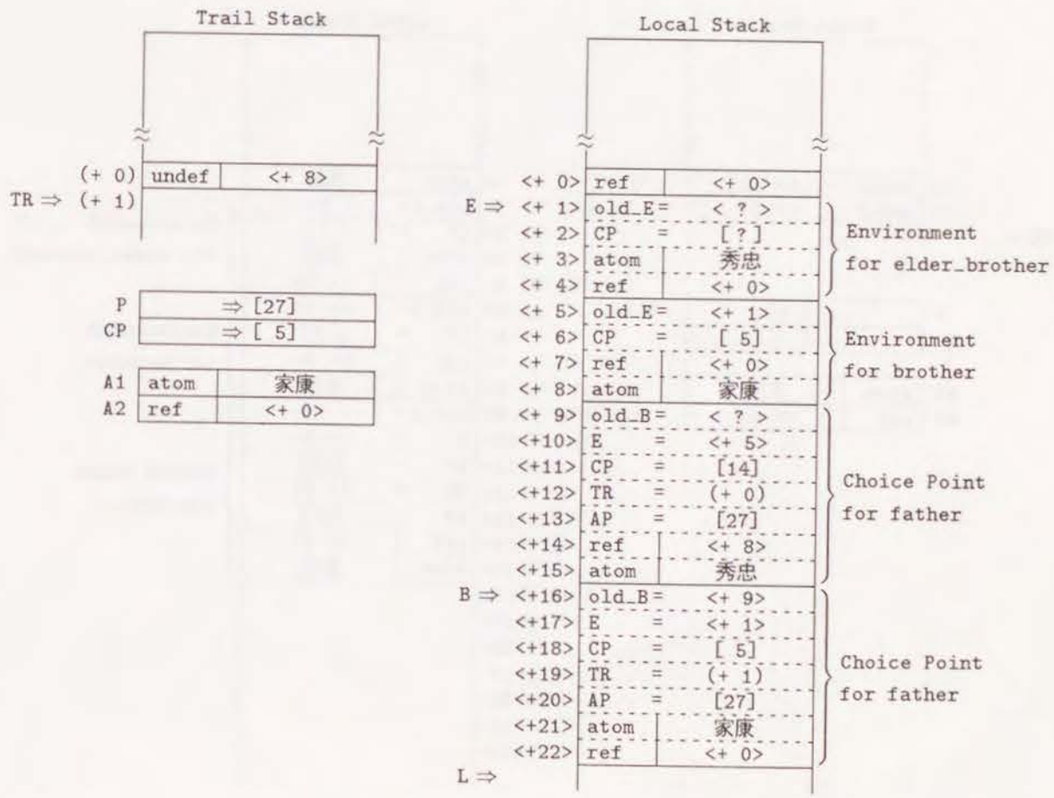


図 D-14: elder の失敗 — 2

(14) father(家康，秀康) の実行

次に，以下に示す述語 father の第三クローズの命令列が実行され，ユニフィケーションがいずれも成功して elder_brother に戻る。

```
[27]  trust_me                        %
[28]  get_constant      家康，A1      % A1=家康，
[29]  get_constant      秀康，A2      % A2=unbound(B)-> 秀康)．
[30]  proceed                         %
```

図 D-15に，この時点での状態を示す。

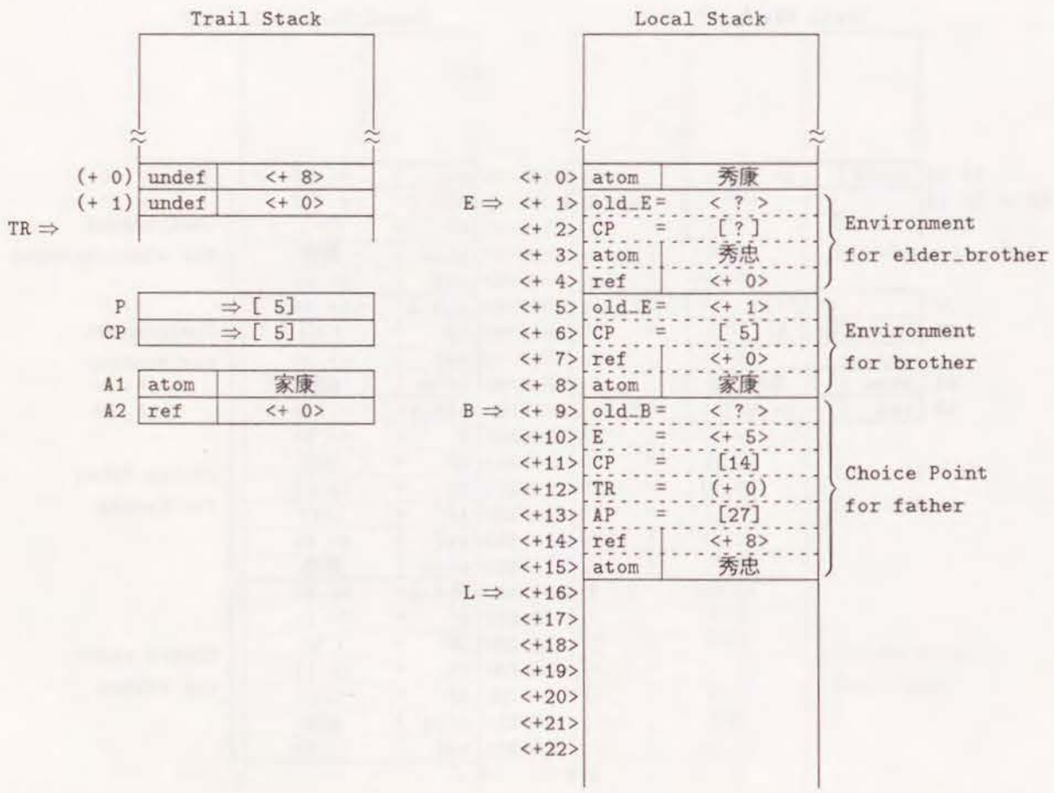


図 D-15: father(家康，秀康) の実行

(15) elder の呼び出し — 3

次に，以下に示す述語 elder_brother の命令列が実行され，elder の三度目の呼び出しが行われる。

```
[ 5]  put_value         Y1，A1      % A1= 秀忠
[ 6]  put_value         Y2，A2      % A2=ref(B)= 秀康
[ 7]  deallocate                    %
[ 8]  execute           elder       %
```

図 D-16に，この時点での状態を示す。

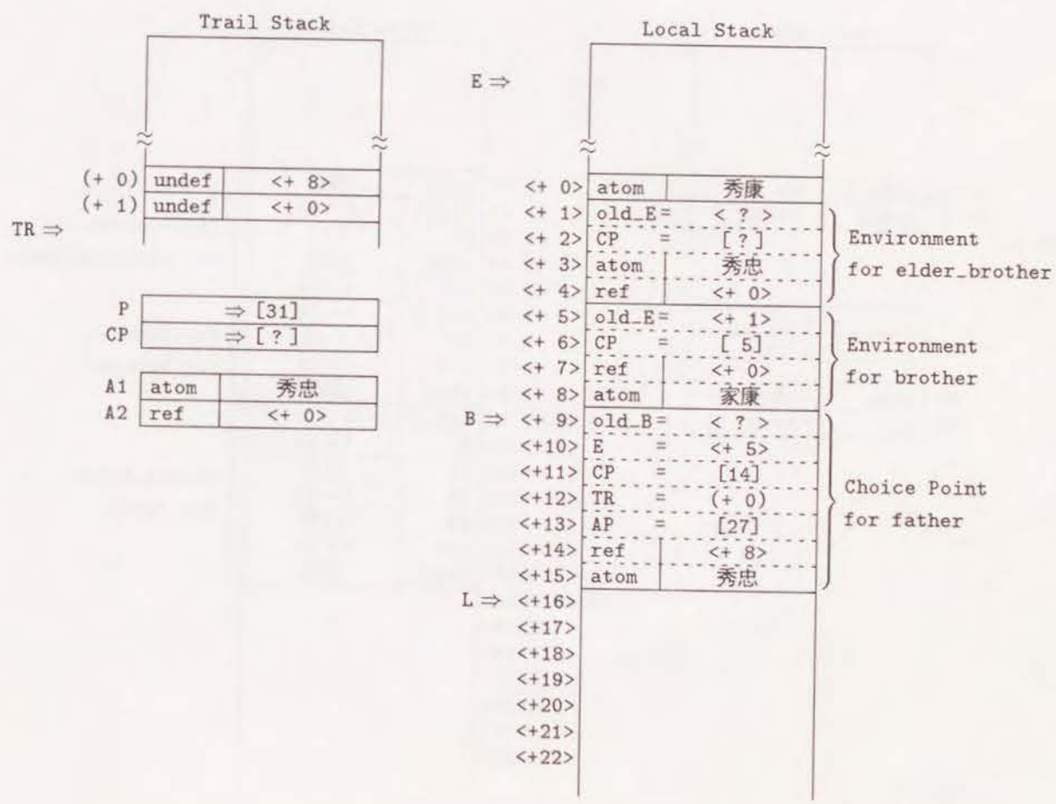


図 D-16: elder の呼び出し — 3

(16) elder の完了

次に，以下に示す述語 elder の命令列が実行され，ユニフィケーションがいずれも成功して，実行が完了する。

```
[31]  switch_on_term     elder_c1, elder_idx, fail, fail
[32]  switch_on_constant     <信康:elder_idx1>, <秀忠:elder_c3a>
[44]  get_constant       秀忠, A1     % A1=秀忠
[45]  get_constant       秀康, A2     % A2=秀秀
[46]  proceed                         %
```

図 D-17に，この時点での状態を示す。

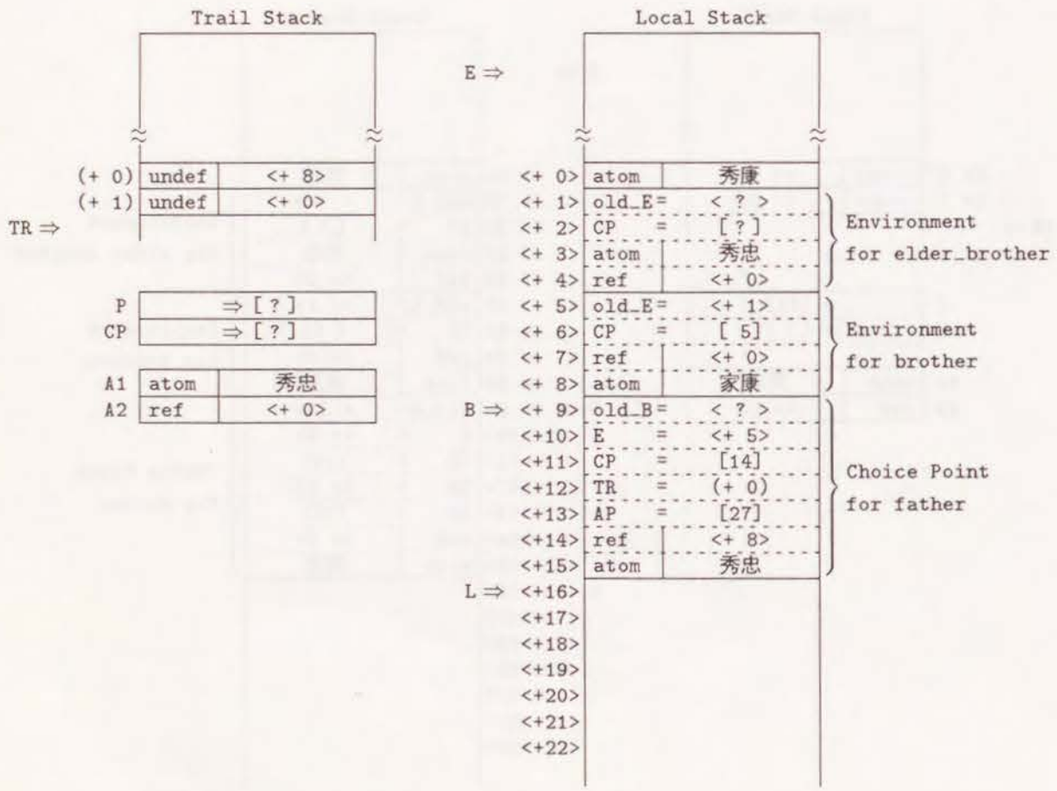


図 D-17: elder の完了